FAULT MANAGEMENT FOR SERVICE-ORIENTED SYSTEMS

by

AMAL ABEDALLA ALHOSBAN

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2013

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

_____

_____

_____

_____

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to my PhD advisor, **Dr. Zaki Malik**, for supporting me during these past years. I could not be prouder of my academic roots and it has been an honor to be Dr. Zaki's first Ph.D. student. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating.

I would also like to thank my PhD committee members Dr. Hongwei Zhang, Dr. Chandan Reddy and Dr. Brahim Medjahed who provided encouraging and constructive feedback. Thank you for helping to shape and guide the direction of the work with your careful and instructive comments.

The completion of my dissertation has been a long journey. I would not have contemplated this road if not for my husband, Issam, who instilled within me a love of creative pursuits and science, all of which finds a place in this dissertation. I owe a lot to Issam, who encouraged and helped me at every stage of my personal and academic life, and longed to see this achievement come true. To Issam, thank you. My daughters, Rana, Salma and Marina, have also been the best along this journey by their understanding, support and endless love. This dissertation would also not be possible without the love and support of my parents, Abedalla and Hind and I dedicate this dissertation to them.

I will forever be thankful to my sisters, brothers and friends for their continuous love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Fault management is defined as *the characteristic by which a distributed system can mask the fault's occurrence and recover from it* (Gadgil, Fox, Pallickara, and Pierce, Gadgil et al.2007). This means that when a system can determine a fault and recover from it, then we can say that it has fault management capability. The main role for fault management is to increase the ability of a system to perform its function correctly even in the presence of internal faults, thus increasing the dependability of a system (Burns and Wellings, Burns and Wellings2001). Two main actions must be performed at any fault occurrence (Denaro, Pezzé, Tosi, and Schilling, Denaro et al.2006): detection and recovery. Fault detection is the first step in the system to assess if a specific functionality is, or will be faulty. After the system has detected a fault, the next step is to prevent or recover from this fault, this is defined as fault recovery. The goal of fault detection is to verify that the services being provided are functioning properly. The simplest way to perform such a task is through observation (e.g., log file) and manual removal of incorrect values. The techniques for detection are the following: self diagnosis, group detection and hierarchical detection. Through self diagnosis the node itself can identify faults in its components. With group detection, several nodes monitor the behavior of other nodes. Finally, in hierarchical detection, fault detection is performed using a detection tree where a hierarchy is defined for the identification of faulty nodes.

There are two general techniques used for developing robust systems against system faults. These are:

1. *Fault tolerance* aims to control either hardware or software faults and continue the system

operation with a reduction in throughput or an increase in latency. The main method used is exception handling, which provides activities to handle various faults (Liu, Li, Huang, and Xiao, Liu et al.2010).

2. *Fault management* provides users with information of existing faults as accurately and informatively as possible, to enable detection of malfunctions of desired properties, and diagnosis of the root causes.

Fault tolerance focuses on improving the availability and reliability of distributed systems. Fault management compliments fault tolerance by enabling users to (1) fix the design or the implementation to strengthen the robustness of distributed systems and (2) detect and analyze malicious behaviors to minimize the impact on the systems.

Current fault management techniques have two typical features: First, they are added a posteriori to existing applications. This means that the applications are not designed for being fault-managed. They often lack a suitable architecture and satisfactory means for the diagnosis and repair of faults. Second, they typically use external management functionality. State information is extracted from the application and analyzed by an external manager, which makes them much more difficult to diagnose and correct (also violating the principle of encapsulation) (Kokash, Kokash2007).

In related literature, fault management is divided into different stages (Gadgil, Fox, Pallickara, and Pierce, Gadgil et al.2007) (Dialani, Miles, Moreau, Roure, and Luck, Dialani et al.2002) (Ardissono, Console, Goy, Petrone, Picardi, Segnan, and Dupré, Ardissono et al.2005) (Demsky and Rinard, Demsky and Rinard2003): (1) *Detection* aims to discover potential faults as early, and

as close to their origin as possible, (2) *Diagnosis* traces the detected fault symptom back to the cause, and (3) *Repair* takes measures to eliminate the fault effects. Similarly, (Paradis and Han, Paradis and Han2007) divides the stages as: (1) *Fault prevention* to avoid a fault, (2) *Fault detection* using different matrices to collect symptoms of possible faults, (3) *Fault isolation* to correct different types of faults received from the networks and proposed hypothesis, (4) *Fault identification* to test each proposed hypothesis and (5) *Fault recovery* to treat from faults. In (Lutfiyya, Bauer, Marshall, and Stokes, Lutfiyya et al.2000), these are: (1) *Fault detection*: Which monitors execution of a distributed system and checks the observations against its expected behaviors. The fault is reported whenever a deviation from the expected behavior is discovered. Instead of manual inspection, automated processes are introduced. (2) *Fault diagnosis*: Once a fault is detected, additional mechanisms are utilized to diagnose the system to identify the nature of the fault and track the root causes. (3) *Evidence Generation*: Evidence can be defined as a set of processed information that demonstrates the assertions drawn from fault diagnosis. In addition, (Katchabaw, Lutfiyya, Marshall, and Bauer, Katchabaw et al.1996) (Wu, Wei, and Huang, Wu et al.2009) (Mulo, Zdun, and Dustdar, Mulo et al.2010) (Halima, Drira, and Jmaiel, Halima et al.2008) proposed that fault management can be divided into: (1) *Fault monitoring*, (2) *Fault diagnosis*, and (3) *Recovery*. However, in (Lyu, Lyu2007) proposed four techniques: (1)*Fault prevention*. (2)*Fault removal*. (3)*Fault tolerance*: by using redundancy. (4)*Fault forecasting*: to estimate the occurrence consequence of the faults. Similarly, (Hanemann, Hanemann2006) (Cheng, Li, and Chen, Cheng et al.2008) proposed: (1) *Detection*, (2) *Diagnosis* and (3) *Recovery*.

A Failure is defined as a condition where the a running system deviates from its specified be-

havior. The cause of a failure is thus called an error, which represents an invalid system state. The error itself is therefore the result of a defect/fault in the system. In other words, a fault is the root cause of a failure, which means that an error is merely the symptom of a fault. A fault may not necessarily result in an error. Similarly, a fault may result in multiple errors. Moreover, a single error may lead to multiple failures, and a fault may be the result of one or more errors (Ben Lakhal, Kobayashi, and Yokota, Ben Lakhal et al.2009). Physical faults include faults that affect hardware, interaction faults includes all external faults, and development faults include faults that occur during development.

Fault management is the set of functions that detect, isolate, and correct faults. As mentioned earlier, a software system fails when it deviates from its specified behavior. A single error may therefore lead to multiple failures. Thus, fault management mainly includes maintaining and examining error logs, acting on error detection notifications, and tracing, identifying and correcting faults. Since faults can occur in different places (e.g., software application, network connection, hardware resources, etc.), traditional fault management tools are not fully equipped to automatically monitor, analyze, and resolve faults in an SOA, where the focus is usually aimed towards enhancing the QoS at run time. Example QoS measurements include (Yu, Liu, Bouguettaya, and Medjahed, Yu et al.2008) (Zarras, Vassiliadis, and Issarny, Zarras et al.2004):

1. *Reliability* measures the ability of a service operation to be executed within the expected time.

2. *Availability* measures the probability that the service operation is operating at any moment and will do the operation of behalf of the users.

3. *Accessibility* measures the degree that the service operation is able to serve the request (i.e. success rate).

4. *Integrity* measures how the service operation maintains the correctness with respect to the source.

5. *Response time* measures the expected delay between the time that the service operation starts and receiver receives the response.

6. *Cost* measures the expense of managing the fault and recover from it.

Note that the above list is not exhaustive, and other QoS attributes can be added.

1.1 Service-oriented Architecture (SOA)

Services involved in an SOA often do not operate under a single processing environment and need to communicate using different protocols over a network. Under such conditions, designing a fault management system that is both efficient and extensible is a challenging task. In essence, SOAs are distributed systems consisting of diverse and discrete software services that work together to perform the required tasks. Reliability of an SOA is thus directly related to the component services' behavior, and sub-optimal performance of any of the components may degrade the SOA's overall quality. The problem is exacerbated due to security, privacy, trust, etc. concerns, since the component services may not share information about their executions. This lack of information translates into traditional fault management tools and techniques not being fully equipped to monitor, analyze, and resolve faults in SOAs.

An SOA is defined as "*a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains*" (Paradis and Han, Paradis and Han2007) (Katchabaw, Lutfiyya, Marshall, and Bauer, Katchabaw et al.1996). In other words, boundaries of SOAs are usually *explicit*, i.e., the services need to communicate across boundaries of different geographical zones, ownerships, trust domains, and operating environments. Moreover, explicit message passing is applied in SOAs instead of implicit method invocations. The services in SOAs are *autonomous*, i.e., they are independently deployed, the topology is *dynamic*, i.e., new services may be introduced without advanced acknowledgement, and the applications consuming a service can leave the system or fail without notification. Services in SOAs *share* schemas and contracts. The message passing structures are specified by schemas, and message-exchange behaviors are specified by contracts. Service compatibility is thus determined based on explicit policy definitions that define service capabilities and requirements.

Two major entities are involved in any SOA transaction: service consumers and service providers (see Figure 1.1). As the name implies, service providers provide a service on the network with the corresponding service description (Malik and Bouguettaya, Malik and Bouguettaya2009) (Lin, Lu, Lai, Chebotko, Fei, Hua, and Fotouhi, Lin et al.2008). A service consumer needs to discover a matching service to perform a desired task among all the services published by different providers (Keromytis, Keromytis2007). In situations where a single service does not suffice, multiple services could be *composed* to deliver the required functionality. Finally, the consumer binds to the newly discovered service(s) for execution, where input parameters are sent to the service provider and output is returned to the consumer (Guinea, Guinea2005) (Denaro, Pezzé, Tosi, and

Schilling, Denaro et al.2006).

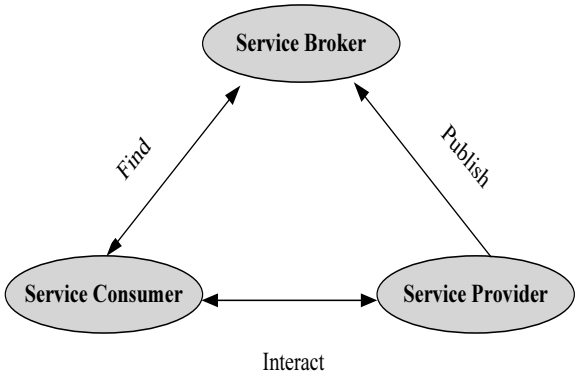


Figure 1.1: Service-Oriented Interaction Model.

A fault may occur at any of the interaction stages. At *publication* stage, faults are normally caused by an incorrect description, which can only be detected by checking the description files. Publishing faults may be related to format deployment or context, and when the format of the description is incorrect, there is an incomplete description. There might be some other faults when a service provides different versions of features than are published in the description. If the description mentions features but those are not provided by the deployed service, these are called Missing Features (Robinson and Kotonya, Robinson and Kotonya2008).Similarly, if the feature described does not match the feature actually provided, it is an Incorrect Feature Description. Service deployment faults occur when the service is not successfully deployed on the target platform. In case the service is missing a Required Resource, the service may be deployed successfully but will fail to perform as desired. Content faults can be detected by validating using predefined criteria. However, it has been reported (Dudley, Joshi, Ogle, Subramanian, and Topol, Dudley et al.2004) that deployment faults cannot be detected before the execution.

The faults during *discovery* may occur either on search invocation or while returning the found services (Gadgil, Fox, Pallickara, and Pierce, Gadgil et al.2007). These are relatively easy to detect if no service is found. On the other hand, a wrong service found fault is difficult to detect. The fault can only be detected when the service is actually invoked or executed. The found service may fail if the client specifies incorrect search criteria, there is a faulty lookup service, or the provided specification does not match the actual provided service(s).

The *composition* process may also present different faults due to various reasons: (1) if the components are incompatible, then the services cannot be connected. (2) Parts of the composition are missing or services are required to translate between services are missing. (3) The returned composition may not meet the specified requirements. (4) Certain properties are not supported by all parts of the composition (e.g. security may only be guaranteed by the first and last service, but not in between). (5) Preconditions, post conditions, or invariants are not fulfilled resulting in the contract between the services being violated. Moreover, logical faults, system faults, content faults, and SLA faults described above may appear in composition at run time (Liu, Li, Huang, and Xiao, Liu et al.2010).

During the *binding* process, the service consumer and service provider negotiate the conditions to execute the service (Hashmi, Alhosban, Malik, and Medjahed, Hashmi et al.2011). The binding may be denied if authorization is denied, authentication fails, or accounting problems occur. Insufficient security may also be a reason (e.g. one side does not trust the certificate of the other) for fault at this stage.

Execution faults occur when the service is executed but the result does not match the expected

outcome. If the service delivers an incorrect result, this can be either due to a software fault or incorrect input (Katchabaw, Lutfiyya, Marshall, and Bauer, Katchabaw et al.1996) (Paradis and Han, Paradis and Han2007). To summarize these faults we have

- Publication stage faults: Incorrect description, service description mismatch, content fault, missing a required resource.
- Discovery stage faults: Required service does not exist or not listed in lookup service, faulty lookup service.
- Composition stage faults: No valid composition, changes in contract.
- Binding stage faults: Authorization denied, authentication failed, accounting problems.
- Execution stage faults: Mismatched results, service crash.

SOAs can be dynamically and flexibly composed by integrating new and existing component services to form complex processes and transactions using standard protocols such as SOAP and WSDL. Each service in an SOA may be invoked using a different invocation model. Here, an invocation refers to triggering a service (by calling the desired function and providing inputs) and receiving the response (return values if any) from the triggered service. An SOA may thus be categorized as a 'composite service', which is a conglomeration of services with invocation relations between them. There are six major invocation relations: Sequential Invocation, Parallel Invocation, Probabilistic Invocation, Circular Invocation, Synchronous Activation, and Asynchronous Activation (D'Mello and Ananthanarayana, D'Mello and Ananthanarayana2009). A brief overview of these follows(see Figure 1.2).

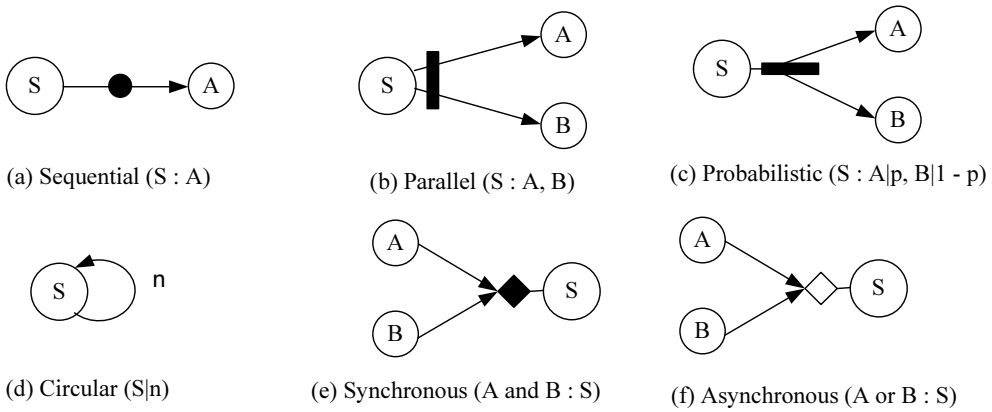


Figure 1.2: Major SOA Invocation Models.

Sequential Invocation: In sequential invocation, a service S invokes its unique succeeding service A (Menasce, Menasce2004). It is denoted as Sequential (S : A) (see Figure 1.2-(a)). Sequential invocation is also defined as a serial invocation (Yu, Zhang, and Lin, Yu et al.2007). A fault may occur in a sequential invocation if the succeeding service fails (service A in Figure 1.2-(a)), or if the connection/link between service S and service A is broken, i.e., service S cannot reach service A. Similarly, if there is no response from one of the services or response times-out from the invoked service (Vaculin, Wiesner, and Sycara, Vaculin et al.2008).

Parallel Invocation: In parallel invocation, a service S invokes its succeeding services in parallel (Menasce, Menasce2004). For example, if S has successors A and B which are independent, S can invoke both A and B at the same time. It is denoted as Parallel (S : A, B) (see Figure 1.2-(b)). Parallel invocation faults may occur if either of service A or service B fails. Since these work in parallel, a fault in one of the services will effect the system model.

Probabilistic Invocation: In probabilistic invocation, a service S invokes its succeeding service(s) with a probability (see Figure 1.2-(c)). For example, if S invokes successor A with the

probability p and successor B with the probability 1 - p, it is denoted as Probabilistic (S : A|p, B|1 - p). The probabilistic invocation is also defined as fork invocation (Menasce, Menasce2004). In this type of invocation, faults may occur when the probability of service A and probability of service B equal to 1, i.e., we cannot invoke service A or service B, or if service A and service B fail, or there are missing links from service S to service A and service B.

Circular Invocation: In circular invocation, a service S invokes itself*n* times. It is denoted as Circular (S|n). A circular invocation can be defined as cloning itself *n* times (see Figure 1.2-(d)). A fault may occur if we encounter an infinite loop, which means that $n = \infty$, or if $n = 0$ which means that this service cannot invoke itself (when it should) (Bai, Hu, Xie, and Ng, Bai et al.2005).

Synchronous Activation: In synchronous activation, a service S is activated only when all its preceding services have been completed. For example, if S has synchronous predecessors A and B, both these services would need to complete before S can progress. It is denoted as Synchronous (A, B : S) (see Figure 1.2-(e)). Faults encountered in synchronous activation are similar to the ones discussed above for parallel invocation.

Asynchronous Activation: In asynchronous activation, a service S is activated as the result of the completion of one of its preceding services (Menasce, Menasce2004). For example, if S has asynchronous predecessors A and B, either A or B's completion would cause S to progress. It is denoted as Asynchronous (A, B : S) (see Figure 1.2-(f)). Asynchronous activation faults are similar to the ones discussed above for the probability invocation model. Table 1.1 shows some of the major faults that may appear in each model.

Self-healing systems have the ability to modify their own behavior in response to changes in

Table 1.1: Expected Faults in Invocation Models

Invocation model	Expected faults
Sequential	Service A failed, missing the link to service A, i.e. service S cannot reach service A, no response, response time-out.
Parallel	Service A failed, service B failed, missing the link to service A or missing link to service B, i.e. service S cannot reach service A and/or cannot reach service B.
Probabilistic	Probability of service A and probability of service B equal to 0, service A failed, service B failed, missing the link to service A or missing link to service B, i.e. service S cannot reach service A and/or service B.
Circular	Infinite loop, n equal to 0.
Synchronous	Service S did not get activated because service A and/or service B did not complete, service A failed and/or service B failed.
Asynchronous	Service S did not get activated because service A and service B did not complete, service A failed and service B failed.

their environment, such as resource variability, user needs and mobility (Lala and Kumar, Lala and Kumar2002) (Chan and Bishop, Chan and Bishop2009) (Paradis and Han, Paradis and Han2007). The lifecycle of self healing systems consists of four major activities as follows: (1) monitoring the system at runtime, (2) planning the changes, (3) deploying the change descriptions and (4) enacting the changes. A self healing system depends on the following requirements for solving any fault(Jacques-Silva, Challenger, Degenaro, Giles, and Wagle, Jacques-Silva et al.2008).

- Adaptability: The system should enable modification of system properties such as structural, topological, dynamic behavioral and interaction aspects.

- Dynamicity: The system should encapsulate the adaptability concerns during runtime. For example, communication integrity and internal state consistency.

- Awareness: The system should support performance monitoring such as state, behavior, correctness and reliability. It should then be able to recognize performance anomalies.

- Observability: The system should enable monitoring of a resulting self healing system's execution environment. The system may not be able to influence changes in its environment, but it may plan changes within itself in response to the environment.

- Autonomy: The system should provide the ability to address the anomalies which are discovered through awareness and observability in the performance of a resulting system and/or its execution environment. Autonomy is achieved by planning, deploying, and enacting the necessary changes.

- Robustness: The system should provide the ability for a resulting system to effectively respond to unforeseen operating conditions. Such conditions may be imposed by the systems external environment, for example malicious attacks and unpredictable behavior of the systems, as well as errors, faults, and failures within the system.

- Distributability: The system should support effective performance of a resulting system in the face of different distribution/deployment profiles.

- Mobility: The system should provide the ability to dynamically change the physical or logical locations of system's constituent elements.

1.2 Challenges

In the previous sections, we list some of the fault management techniques in different distributed systems. However, not all of these techniques are directly applicable to SOA. The characteristics of SOA raise the following problems for fault management:

- It is hard to apply a non-distributed management approach to SOA. A manager needs to communicate with the managed nodes through authentications.

- The management of a SOA is composed of services that may lie outside the management domain of the current system. System management usually involves management of web servers, application servers, databases, and other infrastructures which are usually under a single domain. Service management and infrastructure management should be unified to achieve the goals of SOA fault management.

- Managers and Services may run on heterogeneous platforms and virtual machines where the services are deployed on, and they need to be integrated together.

- A management system needs to adapt to the different QoS and physical service changes.

1.3 Thesis Goals

Many efforts have been made to resolve the faults in distributed systems in general (Castro and Liskov, Castro and Liskov2002) (Zhao, Zhao2007) (Katchabaw, Lutfiyya, Marshall, and Bauer, Katchabaw et al.1996) (Castro, Rodrigues, and Liskov, Castro et al.2003)
(Kotla, Clement, Wong, Alvisi, and Dahlin, Kotla et al.2008) (Singh, Fonseca, Kuznetsov, Rodrigues, and Maniatis, Singh et al.2009) (Aghdaie and Tamir, Aghdaie and Tamir2009) (Merideth, Iyengar, Mikalsen, Rouvellou, and Narasimhan, Merideth et al.2005) (Castro and Liskov, Castro and Liskov2002) (Santos, Lung, and Montez, Santos et al.2005) (Morgan, Shrivastava, Ezhilchelvan, and Little, Morgan et al.1999) (Liang, Lo, Kao, Yuan, and Chang, Liang et al.1997) (Kon-

togiannis, Lewis, Smith, Litoiu, Muller, Schuster, and Stroulia, Kontogiannis et al.2007) (Denaro, Pezzé, Tosi, and Schilling, Denaro et al.2006) (Chan and Bishop, Chan and Bishop2009) (Garlan and Schmerl, Garlan and Schmerl2002) (Paradis and Han, Paradis and Han2007) (Wu, Wei, and Huang, Wu et al.2009) (Griffith, Kaiser, and López, Griffith et al.2009) (Guinea, Guinea2005) (Keromytis, Keromytis2007) (Park, Youn, and Lee, Park et al.2009). The problems in the current fault management mechanisms is that they are not completely suitable for SOAs due to their special needs and features. Thus, we cannot use the distributed system fault management for SOA because of the following reasons:

- Usability: The first challenge involves improving the usability of fault management. Specifically, in the following three aspects: (1) Allowing mechanisms to be applied to applications written in any language (2) Enabling fault management without manual modification of source code (3) Allowing users to specify expectation of system behavior.

- Execution Time: improving the ability of system operators to reason about time. Most mechanisms increase the waiting time which will increase the overall execution time of the system.

- Utilization of Distributed Resources: Most detection and diagnosis mechanisms adopt centralized approaches. However, in SOA we need a decentralized approach.

- Impact Analysis and Repair: Given detected faults, and their root causes, a challenging research topic is how to accurately estimate their impact on the current system and how to repair the system online, i.e. without recompilation and rerun. As a secondary goal it should

also minimize the impact of faults on a running system.

Considering requirements of SOAs, we give the following insights about the objective of the current research for fault management in the context of self-healing for SOA (**FLEX**).

- **FLEX** needs to guarantee high levels of availability, especially for system critical services. Thus, the monitoring component needs improvement. The monitoring process when active will place additional load on the provider, since it should response to other requests generated by the monitor system. Previous studies show that the monitoring increases the system overhead by sending control messages. In our work we reduce this overhead by minimizing these control messages based on the history of the services.

- **FLEX** should be able to detect the fault, and recover from it in a best effort way. The recovery process should have the ability to resolve the fault and put the system in stable state. The selection of how to recover from the fault is a challenge. We combine between the existing processes and find the best solution at run time.

- Fault management in **FLEX** should have a high level of reliability. Ideally speaking we should have a fault free system. **FLEX** should be able to prevent these faults from happening and respond by executing recovery plans preemptively. Previous studies are more focused on the expected faults, which means that the system is analyzed and a database of expected faults is maintained. In our work, we include the detection and prevention/recovery of unexpected faults in the system.

- **FLEX** needs to be automated. In previous studies, the recovery or repair plans are usually

predefined during the design phase. However, for SOA we need on-demand plans that are created at run-time. These plans should be applicable for the current expected or unexpected faults.

- The traditional recovery plans in **FLEX** are time consuming. These may not be suitable for the current services or do not meet consumer's performance requirements. **FLEX** reduces the waiting and processing times as much as possible. It considers the consumer priorities and requirements if a change occurs in the system.

1.4 Motivating Example

In this section, we present an example scenario to motivate the problem and associated solution. Assume a travel planning system that is based on a service-oriented architecture (Figure 1.3.). The company provides travel planning services that include hotel booking, flight reservation, and car rental. In addition to these reservation services, the system also provides an insurance service for the entire trip or individual travel components.

A student (Sam) intends to attend a conference in London, UK. He needs to purchase an airline `ticket` and reserve a `hotel` for this travel. Moreover, he needs some `transportation` to go from the airport to the hotel and from the hotel to other venues (since this is the first time he's visited the UK, he intends to do some "Site-seeing" also). Sam has a restricted budget, so he is looking for a "deal".

Assume that Sam would be using a SOA-based online service (let's call it *SURETY*) that is a one-stop shop providing all the five options (airline ticket, hotel, attractions, transportation and

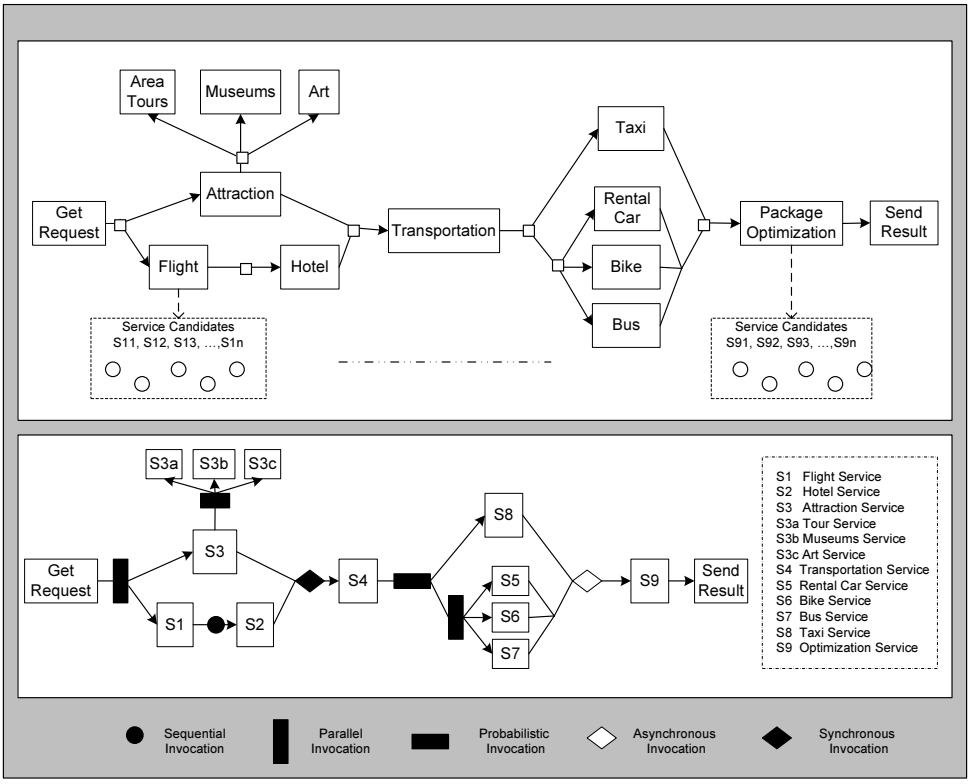


Figure 1.3: Scenario with Invocation Models.

discounts) through outsourcing. *SURETY* provides many services such as: attraction service which outsources to three services (representing individual services): Art, Museums, and Area tours. This service provides arrangement to visit different areas through sub-contractor companies. For clarity, Figure 1.3 shows the options at one level. Sam may select Art, Museum, Area tours, or any combination of these services. In terms of transport options, Sam can either use a taxi service, or move around in a rental car, bus, or bike. The different transport companies provide services based on the distance between the places (attractions, etc.) Sam plans to visit. *SURETY* also provides a package optimization service that finds "deals" for the options chosen by Sam.

In Figure 1.3, the potential services are shown for clarity from "Get request" (when *SURETY*

receives Sam's request) to "Send result" states (when *SURETY* sends result(s) to Sam). This is done to show a combination of different invocation models. In reality, service invocations may not follow such a *flat* structure. Since Sam is looking for a travel arrangement that include: booking a ticket, booking a hotel, transportation (rental car, bike or bus) or taxi, and visiting some places, some of these services can be invoked in parallel (here we assume that *SURETY* provides such an option). Booking a ticket and finding attractions is an example of *parallel invocation*. Among the three choices that Sam can select from (Area tours, Museums, and Art), for area attractions, he has to make a choice among these service instances; this is an example of *probabilistic invocation*. Similarly, taxi or rental car, bike and bus services can be classified as probabilistic invocations since *SURETY* has to invoke one service from among multiple services. *SURETY* then provides the results of transport selection to the Package Optimization service, which hunts for available discounts (e.g., if the customer uses the system for more than one year he will get a 20%, etc.). This invocation is an example of *asynchronous invocation*, as one of the transport selections will suffice. *SURETY* then sends the final selection itinerary to Sam.

In *SURETY*, the system includes multiple services and the fail of one of these service will affect the overall system reliability. For example, if the flight service does not response within a period of time the system will not invoke the hotel service until it solves this issue. Moreover, the hotel service is executed without problems but returns wrong outputs (i.e., reservation for different date). This fault may cased by incorrect input of the client. Another scenario, the transportation service received too many orders which went to out of time or the attraction service denied the binding because of the security certificate. In the following, we predict the previous and other faults and

recover from them.

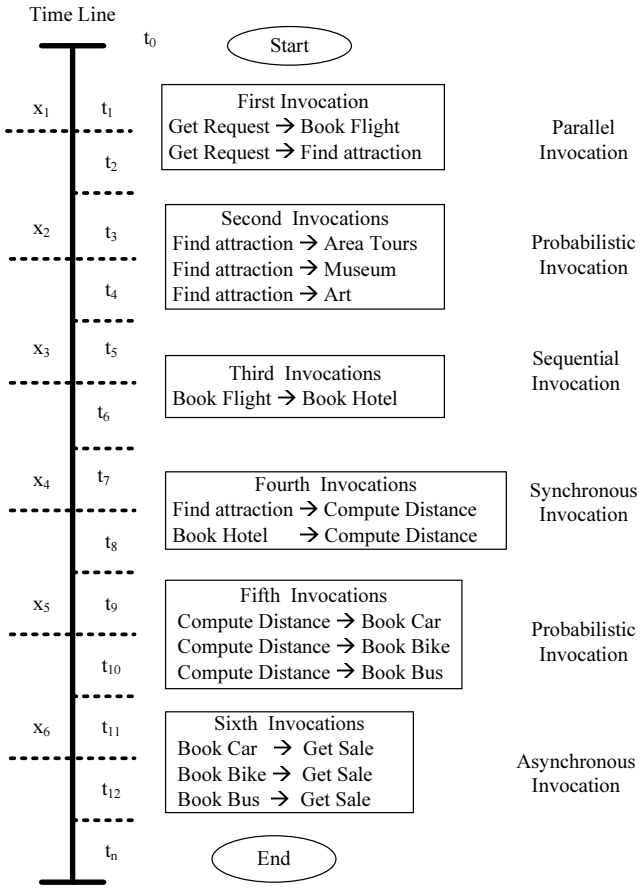


Figure 1.4: Time line for Invocation Models.

In Figure 1.4, we link the invocation models to our scenario through time line. A composed service using one or more of the invocation models described above, may encounter a fault during its execution. The likelihood of encountering a fault is directly proportional to the system's complexity, i.e., the more the invocation models involved, the greater the likelihood of a fault's occurrence.

1.5 Thesis Organization

The organization of this dissertation is as follows. Chapter 2 presents a survey of the related work (i.e., literature review) . Chapter 3 briefly explains the motivation for the work our proposed self-healing framework. Chapter 4 presents the proposed semantic similarity and ranking technique. Chapter 5 presents our fault management propagation approach. Chapter 6 presents the performance analysis for the proposed techniques. Chapter 7 concludes the dissertation and presents possible directions for future research work.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we provide a brief overview of the related literature for each stage of our proposed framework. Thus, we compared FOLT with similar prediction techniques; compared S^2R with other matching techniques; and we compared FLEX with multiple fault management techniques that use planning.

Current fault management techniques have two typical features: First, they are added a posteriori to existing applications. This means that the applications are not designed for being fault managed. They often lack a suitable architecture and efficacious means for the diagnosis and repair of faults. Second, they typically use external management functionality. State information is extracted from the application and analyzed by an external manager, which makes them much more difficult to diagnose and correct. This also violates the principle of encapsulation (Kokash, Kokash2007).

Developing fault tolerance mechanisms for distributed systems will become less difficult if they are provided with manageability. A system's manageability refers to its capability to be managed, how much a service can report, and change the system's state, and the ease of interaction with the service. The manageability of a system depends on the manageability of its subsystems and components. It is impossible to manage a large-scale SOA if services offer no manageability. Developing services with built-in manageability is indispensable for composing manageable SOAs. The manageability of a service depends on the manageability features of its platform and its design.

The researchers divide fault management technique into different stages. For instance, fault management is divided into three phases in (Gadgil, Fox, Pallickara, and Pierce, Gadgil et al.2007) (Dialani, Miles, Moreau, Roure, and Luck, Dialani et al.2002) (Ardissono, Console, Goy, Petrone, Picardi, Segnan, and Dupré, Ardissono et al.2005) (Demsky and Rinard, Demsky and Rinard2003): (1) *Detection* aims to discover potential faults as early, and as close to their origin as possible, (2) *Diagnosis* traces the detected fault symptom back to the cause, and (3) *Repair* takes measures to eliminate the fault effects. Similarly, (Paradis and Han, Paradis and Han2007) divides the stages as: (1) *Fault prevention* to avoid a fault, (2) *Fault detection* using different matrices to collect symptoms of possible faults, (3) *Fault isolation* to correct different types of faults received from the networks and proposed hypothesis, (4) *Fault identification* to test each proposed hypothesis and (5) *Fault recovery* to treat from faults. In (Lutfiyya, Bauer, Marshall, and Stokes, Lutfiyya et al.2000), these are: (1) *Fault detection*: Which monitors execution of a distributed system and checks the observations against its expected behaviors. The fault is reported whenever a deviation from the expected behavior is discovered. Instead of manual inspection, automated processes are introduced. (2) *Fault diagnosis*: Once a fault is detected, additional mechanisms are utilized to diagnose the system to identify the nature of the fault and track the root causes. (3) *Evidence Generation*: Evidence can be defined as a set of processed information that demonstrates the assertions drawn from fault diagnosis. In addition, (Katchabaw, Lutfiyya, Marshall, and Bauer, Katchabaw et al.1996) (Wu, Wei, and Huang, Wu et al.2009) (Mulo, Zdun, and Dustdar, Mulo et al.2010) proposed that fault management can be divided into: (1) *Fault monitoring*, (2) *Fault diagnosis*, and (3) *Recovery*. However, in (Lyu, Lyu2007) proposed four techniques: (1)*Fault prevention*.

(2)*Fault removal*. (3)*Fault tolerance*: by using redundancy. (4)*Fault forecasting*: to estimate the occurrence consequence of the faults. Similarly, (Hanemann, Hanemann2006) (Cheng, Li, and Chen, Cheng et al.2008) proposed: (1) *Detection*, (2) *Diagnosis* and (3) *Recovery*. Table 2.1 lists some major related works and the techniques used there in.

Table 2.1: Fault Management Techniques in Literature.

Technique	(Gadgil, Fox, Pallickara, and Pierce, Gadgil et al.2007) (Dialani, Miles, Moreau, Roure, and Luck, Dialani et al.2002) (Ardissono, Console, Goy, Petrone, Picardi, Segnan, and Dupré, Ardissono et al.2005) (Demsky and Rinard, Demsky and Rinard2003)	(Katchabaw, Lutfiyya, Marshall, and Bauer, Katchabaw et al.1996) (Wu, Wei, and Huang, Wu et al.2009) (Mulo, Zdun, and Dustdar, Mulo et al.2010)	(Lutfiyya, Bauer, Marshall, and Stokes, Lutfiyya et al.2000)	(Laster and Olatunji, Laster and Olatunji2007) (Huang, Zou, Wang, and Cheng, Huang et al.2005) (Maurel, Diaconescu, and Lalanda, Maurel et al.2010)	(Hanemann, Hanemann2006) (Cheng, Li, and Chen, Cheng et al.2008) (Santos, Lung, and Montez, Santos et al.2005)	(Paradis and Han, Paradis and Han2007)
Prevention	N	N	N	N	N	Y
Detection	Y	N	Y	N	Y	Y
Monitoring	N	Y	N	Y	N	N
Diagnosis	Y	Y	Y	Y	Y	N
Identification	N	N	N	N	N	Y
Planning	N	N	N	Y	N	N
Repair	Y	N	N	N	N	N
Recovery	N	Y	N	Y	Y	Y

2.1 Traditional Fault Management Strategies

There is a growing demand for highly-available systems that provide correct service without interruptions. These systems must manage the faults because they are a major cause of outages. Fault management is divided into: hardware and software fault management. Hardware fault management is about building computers that automatically recover from faults that occur in hardware components. The techniques employed to do this generally involve partitioning a computing system into modules, each module is backed up with protective redundancy so that, if the module

fails, others can assume its function. Special mechanisms are added to detect faults and implement recovery. Two general approaches to hardware fault recovery have been used, fault masking and dynamic recovery (Singh, Fonseca, Kuznetsov, Rodrigues, and Maniatis, Singh et al.2009) (Kotla, Clement, Wong, Alvisi, and Dahlin, Kotla et al.2008). Fault masking is a structural redundancy strategy that completely masks faults within a set of redundant modules. A number of identical modules execute the same functions, and their outputs are voted to remove fault created by a faulty module. Dynamic recovery is generally more hardware efficient; it is the approach of choice in resource-constrained systems. Its disadvantage is that computational delays occur during fault recovery, fault coverage is often lower, and specialized operating systems may be required. Software fault management is building software that can manage software design faults which are a result of programming errors. There is an approach called design diversity which combines hardware and software fault management and the goal from this approach is to tolerate both hardware and software design faults; however this is a very expensive technique because every detail have to be determined in the design.

Software fault management provides service complying with the relevant specification of faults by using the following: First, single version software techniques, such as monitoring techniques, decision verification, and exception handling are used to partially manage software design faults. Second, multiple version software techniques such as recovery block (RcB), N-version programming(NVP) and N self checking programming(NSCP). Third, the multiple data representation environment utilizes different representations of input data to provide tolerance to software design faults, such as retry blocks (RtB), N-copy programming (NCP) and N-self checking programming.

It has been found that redundancy alone is not sufficient for management of software design faults, so some forms of diversity must accompany the redundancy and combine it with diversity. The goal of diversity is to make the modules as diverse and independent as possible to minimize the identical fault causes. Diversity divided into: design diversity and data diversity.

Diversity can be applied to several layers of the system such as hardware, application, software system operators, and the interfaces between these components (e.g., retry, rollback, roll forward, recovery with check pointing, restart, and hardware reboot). Since exact copies of software component redundancy cannot increase reliability, we need to provide diversity in the design and implementation of the software.

BFT: Byzantine Fault Tolerance

When designing replication protocols, we have to determine the types of faults the protocol is designed to manage. The choice lies between crash fault models, where it is assumed nodes fail cleanly by becoming completely inoperable, or a byzantine fault model, where no assumptions are made about faulty components (Singh, Fonseca, Kuznetsov, Rodrigues, and Maniatis, Singh et al.2009). Byzantine fault tolerance (BFT) allows a replicated service to tolerate arbitrary behavior from faulty replica behavior. To more easily identify the problem, definitions of byzantine fault and byzantine failure are as follows: byzantine fault is a fault presenting different symptoms to different observers and byzantine failure is the loss of system service due to a byzantine fault (Katchabaw, Lutfiyya, Marshall, and Bauer, Katchabaw et al.1996). BFT has low overhead storage and provides good performance and strong correctness guarantees if no more than one-third of the replicas fail. However, it requires all replicas to run the same service implementation

and to update their state in a deterministic way. Therefore, it cannot tolerate deterministic software errors that cause all replicas to fail concurrently, and it complicates the reuse of existing service implementations because it requires extensive modifications to ensure identical values for the state of each replica. The basic idea in BFT is simple. Clients send requests to execute operations, and all no faulty replicas execute the same operations in the same order. Since replicas are deterministic and start in the same state, all no faulty replicas send replies with identical results for each operation. The client chooses the result that appears in at least f+1 replies. The hard problem is ensuring no faulty replicas execute the same requests in the same order. BFT uses a combination of primary backup and replication techniques to order requests. Replicas move through a succession of numbered configurations called views. In a view, one replica is the primary, and the others are backups. The primary picks the execution order by proposing a sequence number for each request. Since the primary may be faulty, the backups check the sequence numbers and trigger view changes to select a new primary when it appears that the current one has failed.

BASE: BFT with Abstract Specification Encapsulation

Fault management using replication is expensive to deploy. BFT with Abstract Specification Encapsulation (BASE) combines BFT with work on data abstraction. The main idea of this combination is to reduce the cost of BFT and improve the performance (Castro, Rodrigues, and Liskov, Castro et al.2003). The goal of BASE is to build a replicated system by reusing a set of off-the-shelf implementations of some service. The BASE methodology corrects BFT problems; they enable replicas to run different implementations. The methodology is based on the concepts of abstract specification and abstraction function from work on data abstraction. BASE offers several

important advantages over BFT: (1) Reuse of existing code. (2) It has been observed that there is a correlation between the length of time software runs and the probability that it fail. BASE combines proactive recovery with abstraction to counter this problem. When a replica is recovered, it is rebooted and restarted from a clean state. Then it is brought up to date using a correct copy of the abstract state. (3) Opportunistic N-version programming. Replication is not useful when there is a strong positive correlation between the failure probabilities of the different replicas. As we mention in the previous sections, N-version programming exploits design diversity to reduce the probability of correlated failures, but it has several problems: it increases development and maintenance costs by a factor of N or more, adds unacceptable time delays to the implementation, and does not provide a mechanism to repair faulty replicas. BASE enables an opportunistic form of N-version programming by taking advantage from off-the-shelf implementations of common services (Zhao, Zhao2007).

Zyzzyva: Speculative Byzantine Fault Tolerance

Zyzzyva is a state machine replication protocol proposed by (Kotla, Clement, Wong, Alvisi, and Dahlin, Kotla et al.2008) and based on three sub-protocols: (1) agreement to order requests for execution by the replicas, (2) view change to coordinate the election of a new primary when the current primary is faulty or the system is running slowly, and (3) checkpoint to limit the state that must be stored by replicas and reduces the cost of performing view changes. Zyzzyva uses speculation to reduce the BFT cost and simplify the design of replication. Unlike in traditional replication where the client orders a request to the replicas, Zyzzyva replicas speculatively execute requests without running an expensive agreement protocol to establish the order. As a result, correct replica

states may diverge, and replicas may send different responses to clients. If a speculative reply and history are stable, the client uses the reply. Otherwise, the client waits until the system converges on a stable reply and history. The challenge in Zyzzyva is ensuring that responses to correct clients become stable because replicas are responsible for ensuring that all requests from a correct client are eventually complete, but a client waiting for a reply and history to become stable can speed the process by supplying information that will either cause the request to become stable rapidly.

Zeno: Eventually Consistent Byzantine-Fault Tolerance

The building of Zeno did not start from scratch but instead it is the adaptation of Zyzzyva (Singh, Fonseca, Kuznetsov, Rodrigues, and Maniatis, Singh et al.2009). Zeno specifies safety and liveness properties of a generic eventually consistent BFT service. Safe, consistent system behaves like a centralized server whose service state can be modeled as a multi-set. Each element of the multi-set is a history which means that a totally ordered subset of the invoked operations being aware of each other, also limits the total number of divergent histories, which in the case of Zeno cannot exceed, at any time,

There are two types of operations, weak and strong. A weak operation may return, with the corresponding result reflecting the execution of all the operations that precede it. In this case, we say that the operation is weakly complete. For strong operations, they must wait until they are committed, which means each history has a prefix related by containment before they can return with a similar way of computing the result. Assuming that each correct client is well-formed, it never issues a new request before its previous (weak or strong) request is (weakly or strongly, respectively) complete. Zeno service guarantees that a request issued by a correct client

is processed and a response is returned to the client, provided that the client can communicate with enough replicas in a timely manner. Zeno is a BFT state machine replication protocol. It requires N = (3 f + 1) replicas to tolerate f Byzantine faults. Zeno has three components: sequence number assignment, to determine the total order of operations; view changes, to deal with leader replica election; and check pointing, to deal with garbage collection of protocol and application state (see Table 2.2).

Table 2.2: Comparison between Zeno and Zyzzyva

Zeno	Zyzzyva
Allows lower overhead	Allows higher overhead
lower latency	Higher latency
Requires clients to use sequential timestamps	Not necessarily sequential timestamps
Disables a single-phase performance optimization	Offers a single-phase performance optimization
Clients send the request to all replicas	Clients send the request only to the primary replica.

CoRAL: Connection Replication and Application-level Logging

Most of the previous methods require deterministic servers or changes to the clients, however, CoRAL recovers in-progress requests and does not require deterministic servers (Aghdaie and Tamir, Aghdaie and Tamir2009). The basic idea of CoRAL is to use a combination of active replication and logging. In the normal scenario of sending and receiving data between client and server, the client sends a request to replicate servers, then the servers send back acknowledgment, after that the servers send the response and the client sends back acknowledgments. However, in CoRAL, requests and replies as usual but does not process requests unless the primary server fails. In addition there is a backup server who receives the request from the client and sends it

to the primary replica, then the primary replica sends the acknowledgment to the client. On the other hand, after receiving the acknowledgement from the primary replica, the client send back the acknowledgment to the backup server which forward this acknowledgement to the primary replica. Through all these steps the backup logs each request while the replica process the request and reply to the client after sending a complete copy to the backup. Internally, the primary replica reply to the backup server if there is no fault, but in case of the fault occur before sending the reply to the client, the backup transmits its copy. If the primary fails before logging the reply in the backup server, the backup processes its copy of the request, generates a reply, and sends it to the client.

Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications

Thema combines between BFT and web services to provide a structured way to build BFT survivable web services that application developers can use like other web services (Merideth, Iyengar, Mikalsen, Rouvellou, and Narasimhan, Merideth et al.2005). BFT assume client-server model without requiring information from other services, in this case, these systems do not provide support for multi-tiered applications that have heterogeneous reliability requirements. However, thema supports the multi-tiered requirements of web services and provide standarized of web services. Thema includes three libraries:(1) a client library that allows client access to BFT web services, (2) a server library to facilitate the creation of these services, and (3) an external service library that allows an external web services to be accessed safely by a BFT. Thema is designed to address the challenges of creating distributed applications which composed of multiple web services such as: works in a mixed fault model using BFT, allows BFT to make safe interaction with external services and provides a support for BFT. In web services to be able to use FTB, it

must support access from non-replicated client and support access to non-replicated web services with internal consistency which is solved by Thema (Merideth, Iyengar, Mikalsen, Rouvellou, and Narasimhan, Merideth et al.2005).

CLBFT: Castro-Liskov Practical Byzantine Fault Tolerance

CLBFT is a BFT state machine replication protocol and library implementation. The purpose of creating this protocol is to create client-server fault tolerant applications. There are two characteristic of CLBFT over BFT: (1) There is no assumptions for safety such as upper bound on communication latency. (2) CLBFT has a better performance than BFT because it does not require the use of public key cryptography during normal operation (Castro and Liskov, Castro and Liskov2002). From the client side, when the client sends a request, the CLBFT library sends the request to the CLBFT service and wait for (f+1) response from different replicas, then ensure that the replicas result is correct. From the server side, the replicas act as independent services but provide the same operations which needs at least (2f+1) replicas to execute. In CLBFT Not all replicas will execute, there are out of date replicas. CLBFT does not provide a mechanism for a replicated service to access external services consistently. This challenge solved in Thema (Castro and Liskov, Castro and Liskov2002).

CORBA: Common Object Request Broker Architecture

This standard defines a set of services for the implementation of replication techniques in distributed environment. CORBA is based on Object Management Group's (OMG's). CORBA has multiple service objects that provide the functionalities for building fault distributed application

such as the following: (1) replication management service is contact the object group management service. (2) group management service which acting dynamically in the input and output of the replicated objects. (3) generic factory is interacting with local factory which is responsible for defining and removing replicas. (4) the property management service is responsible for defining the prosperities of the fault tolerance for each group. (5) the fault management service performs the interfaces of the fault monitoring. In CORBA, fault detection is incorporated at in the server level, object level and process level.(6)recovery and logging service, the main objective of recovery and logging in CORBA is to register requests received by the server and that to keep the replicas in consistent state (Santos, Lung, and Montez, Santos et al.2005) (Morgan, Shrivastava, Ezhilchelvan, and Little, Morgan et al.1999) (Liang, Lo, Kao, Yuan, and Chang, Liang et al.1997).

WS-Replication: Web Service Replication

WS-Replication uses a clustering-based approach to guarantee the availability of the system. Availability is achieved by deploying the same service in a set of sites, so if one site fails, the other continues providing the service. WS-Replication is based on a group communication web service and avoids the use of ad hoc mechanisms. The group communication is a web service called (WS-Multicast) which uses SOAP as a transport protocol. WS-Multicast provides multicast and the notion of views. A view contains currently connected and active members. The processes start when multicast messages are sent to a group, and the system ensures that all available members deliver the same messages. Also, the system ensures that a message that is delivered to all available members even if a member fails (Salas, Perez-Sorrosal, Pati and Jiménez-Peris, Salas et al.2006).

2.2 Fault Prediction

In this section, we provide a brief overview of related literature on fault management and fault tolerance techniques in service-oriented environments, and the Web in general. Santos et al. (Santos, Lung, and Montez, Santos et al.2005) proposed a fault tolerance approach (FTWeb) that relies on active replicas. FTWeb uses a sequencer approach to group the different replicas in order. It aims at finding fault free replica(s) for delegating the receiving, execution and request replies to them. FTWeb is based on the WSDispatcher engine, which contains components responsible for: creating fault free service groups, detecting faults, recovering from faults, establish a voting mechanism for replica selection, and invoking the service replicas. Raz et al. (Raz, Koopman, and Shaw, Raz et al.2002) present a semantic anomaly detection technique for SOAs. When a fault occurs, it is corrected by comparing the application state to three copies of the service code and data that is injected at a host upon its arrival. Similarly, Hwang et al. (Hwang, Wang, Tang, and Srivastava, Hwang et al.2007) analyze the different QoS attributes of web services through a probability based model. The challenge in this approach is composing an alternate work flow in a large search space (withe the least error). Online monitoring (for QoS attributes) also needs some investigation in this approach.

Wang et al's. (Wang, Bandara, and Pahl, Wang et al.2009) approach integrates handling of business constraint violations with runtime environment faults for dynamic service composition. The approach is divided into three phases. The first phase is defining the fault taxonomy by dividing the faults into four groups (functional context fault, QoS context fault, domain context fault and platform context fault) and analyzing the fault to determine a remedial strategy. The second

phase is defining remedial strategies (remedies are selected and applied dynamically). The remedial strategies are categorized into goal-preserving strategies to recover from faults (ignore, retry, replace and recompose) and non-goal preserving strategies to support the system with actions to assist possible future faults (log, alert and suspend). The third phase is matching each fault category with remedial strategies based on the data levels. The main challenge in this approach is the extra overhead, especially when the selected strategy is a "recomposition" of the whole system.

Simmonds et al. (Simmonds, Gan, Chechik, Nejati, O'Farrell, Litani, and Waterhouse, Simmonds et al.2009) present a framework that guarantees safety and aliveness through the conversation between patterns, and checking their behaviors. The framework is divided in two parts: (1) Websphere runtime monitoring with property manager and monitoring manager. The property manger consists of graphical tools to transfer the sequential diagram to NFAs and check the XML file. The monitoring manager builds the automata and processes the events. (2) Websphere runtime engine. It uses the built-in service component that already exists in BPEL, to provide service information at runtime. Delivering reliable service compositions over unreliable services is a challenging problem. Liu et al. (Liu, Li, Huang, and Xiao, Liu et al.2009) proposed a hybrid fault-tolerant mechanism (FACTS) that combines exception handling and transaction techniques to improve the reliability of composite services.

2.3 Dynamic Planning

In this section, we provide a brief overview of some of the related fault recovery approaches. The methods proposed in these works generate recovery plans for SOAs and Web services. In

SOAs, a business process can terminate successfully if all services finish their work correctly (providing alternative plans in case of a fault). One option to recover from the fault is to retry (i.e., Re-do) the failed service. Retry is the easiest to recover from faults since it does not require extra work such as looking for similar Web services. In (Tan, Fong, and Bobroff, Tan et al.2010) a new execution model called BPEL4JOB is proposed. BPEL4JOB designed a fault-handling policy (retry) that uses a signal to indicate the job execution state and adds a retry mechanism for faulty service, until a valid response is received. Another method using retry mechanism was described in (Modafferi, Mussi, and Pernici, Modafferi et al.2006). In the proposed approach a designer defines a WS-BPEL process annotated with information about recovery actions such as retry and then a preprocessing phase, starting from this annotated WS-BPEL, generates a standard WS-BPEL file. In (Lakhal, Kobayashi, and Yokota, Lakhal et al.2006), authors used definition rules, compatibility rules and ordering rules to build a flexible system model. In the proposed model called (WSC), users define a compensating procedure that is invoked in case a fault occurs during an activities' execution life span. The model defined a vitality degree indicating that some activities are identified as optional. The optional activity could be ignored which is a good example of the ignore recovery strategy.

In (Dai, Yang, and Zhang, Dai et al.2009), authors proposed a method (SMP) to predict the QoS and performance of composed and alternative Web services in case of a fault (i.e., replace). The proposed method used a semi-markov model to predict the service data (e.g., execution time, input, output, etc) while the services are running. SMP is the extension of markov process based on time-dependent stochastic behaviors. The proposed method is similar to markov model except

that its probabilities depend on the amount of time since the last transition. The predicted performance of Web services is used to trigger reselection process for alternate Web services used for replacing Web services. The challenge in this method is that it is applied to only one of the metrics for QoS attribute and assumes equivalence of services in term of functionality. Similar strategy (replace) is used in (Chafle, Dasgupta, Kumar, Mittal, and Srivastava, Chafle et al.2006). It proposes a method for adaptation in Web services composition and execution by using multiple workflows and given feedback mechanism between the composition, deployment and runtime stages along with ranking functions. The method selects services based on similarity using the hamming distance function given the QoS dimensions that are handled by the feedbacks coming from different stages. DISC (Zahoor, Perrin, and Godart, Zahoor et al.2010) provides a constraint based declarative approach that allows users to design the composition by identifying and providing a set of constraints that mark the boundary of the solution.

In (Vaculin, Wiesner, and Sycara, Vaculin et al.2008) a retry recovery mechanism is used as CV-handler. The CV-handler presents an approach for specification of exception handling and recovery of semantic web services based on OWL-S. The technique uses standard fault handlers and compensation known from WS-BPEL to provide support for long running transactions. CV-handlers allow a designer to define what situations are supposed to trigger an erroneous state. FTWeb (Santos, Lung, and Montez, Santos et al.2005) is a fault tolerance approach, that relies on active replicas. FTWeb uses a sequencer approach to group different replicas. It aims at finding fault free replica(s) for delegating the receiving, execution and request replies. FTWeb is based on the WS-Dispatcher engine, which contains components responsible for: creating fault free service groups,

detecting faults, recovering from faults, establish a voting mechanism for replica selection, and invoking the service replicas. Many other replication techniques have been proposed such as (Zhao, Zhang, and Chai, Zhao et al.2009). The proposed technique is presented as a lightweight fault tolerance framework for Web services (LFT). In this framework, a Web service can be rendered fault tolerantly by replicating it across several nodes. A consensus-based algorithm is used to ensure total ordering of the requests to the replicated Web services, and to ensure consistent membership view among the replicas. Another example of using replication strategy is WS-Replication (Salas, Perez-Sorrosal, Pati and Jiménez-Peris, Salas et al.2006). The proposed framework provides an infrastructure for WAN replication of Web services. The infrastructure is based on a group communication Web service. Liu et al. (Liu, Li, Huang, and Xiao, Liu et al.2009) proposed a hybrid fault-tolerant mechanism (FACTS) that combines exception handling and transaction techniques to improve the reliability of composite services. Table 2.3 summarizes our findings. A ✓ in a cell means that the corresponding technique provides explicit support for the corresponding recovery strategy, whereas an X indicates that the strategy is not supported.

2.4 Semantic Similarity

In this section, we provide a brief overview of some of the related literature. Several methods have been proposed to deal with the Web service matching problem. The technique in (Xia and Yoshida, Xia and Yoshida2007) uses two stage assessment. In the first stage all service belonging to a specific category are gathered. The second stage consists of finding similarity among these services based on input, output, conditions and effects. LARKS (Sycara, Klusch, Widoff,

Table 2.3: Recovery Planning Techniques Summarized

Technique	Replicate	Replace	Retry	Ignore	Random
SMP (Dai, Yang, and Zhang, Dai et al.2009)	x	✓	x	x	x
BPEL4JOB (Tan, Fong, and Bobroff, Tan et al.2010)	x	x	✓	x	x
WS-BPEL (Modafferi, Mussi, and Pernici, Modafferi et al.2006)	x	x	✓	x	x
WFlow (Chafle, Dasgupta, Kumar, Mittal, and Srivastava, Chafle et al.2006)	x	✓	x	x	x
CV-handler (Vaculin, Wiesner, and Sycara, Vaculin et al.2008)	x	x	✓	x	x
FTWeb (Santos, Lung, and Montez, Santos et al.2005)	✓	x	x	x	x
FACT (Liu, Li, Huang, and Xiao, Liu et al.2009)	x	✓	✓	✓	✓
WS-Replication (Salas, Perez-Sorrosal, Pati and Jiménez-Peris, Salas et al.2006)	✓	x	x	x	x
LFT (Zhao, Zhang, and Chai, Zhao et al.2009)	✓	x	x	x	x
WSC (Lakhal, Kobayashi, and Yokota, Lakhal et al.2006)	x	x	x	✓	x

and Lu, Sycara et al.1999) defines five techniques for service matchmaking: context matching, profile comparison, similarity matching, signature matching, and constraint matching. Matching services to requests is performed by using any combination of the above techniques. The ATLAS matchmaker (Paolucci and Wagner, Paolucci and Wagner2006) defines two methods for comparing service capabilities described in DAML-S. The first method compares functional attributes to check whether advertisements support the required type of service or if it delivers sufficient quality of service. The second method compares the functional capabilities of Web services in terms of inputs and outputs. Anamika (Chakraborty, Perich, Joshi, Finin, and Yesha, Chakraborty et al.2002) presents a service matching technique for pervasive computing environments. Service descriptions are provided in DAMLS. They also include platform specific information such as processor type, speed, and memory availability. The composition manager uses a semantic service discovery

mechanism to select participant services. RACER (Li and Horrocks, Li and Horrocks2003) adopts techniques from knowledge representation to match DAML-S service capabilities. In particular, it defines a description logic (DL) reasoner; advertisements and requests are represented in DL notations.

Another DAML-S based matchmaker implementation is KarmaSIM (Narayanan and McIlraith, Narayanan and McIlraith2002) where DAML-S descriptions are described in terms of a first-order logic language (predicates) and then converted to Petri-nets where the composition can be simulated, evaluated and performed. Context-based matching (CBM) has been proposed in (Medjahed and Atif, Medjahed and Atif2007), the matching process is performed via peer-to-peer interactions between a context-based matching engine, CPAs and community services. A service consumer sends a matching request to context-based matching engine which sends a sub request to the communities and compares the consumer requirement with each community members. Then the context-based matching engine finds the intersection between the matching set from each community. The communities have been created based on the policies inside the Web services. The problem in this technique is that the number of comparisons will be high if the same Web service exists in all communities (i.e., the Web service includes all policies that the consumer has requested). Circular context-based (CCB) has been proposed in (Segev, Segev2008), the technique compares context extracted from each Web service based on its WSDL description to with other Web services' textual description context. The second stage consists of finding the context overlap among the Web service through parsing WSDL file. Other service matching techniques are also presented in (Baïna, Benali, and Godart, Baïna et al.2001) (Heuvel, Yang, and Papazoglou, Heuvel

et al.2001) (Mecella, Pernici, and Craca, Mecella et al.2001). However, these techniques mostly focus on syntactic comparison among attributes of Web services.

Since we use contextual information of Web services, we position our work with existing context-oriented Web service frameworks. Several context-aware approaches have recently been proposed to enhance Web service discovery and composition mechanisms. Context attributes (Lee and Helal, Lee and Helal2003) proposes a context-aware service discovery technique for mobile environments. It defines the context of a Web service as a set of attributes included in the service description. Examples of context attributes include user location and network bandwidth. The discovery engine first lookups for Web services based on traditional criteria (e.g., service category in UDDI). Then, it reduces the qualified services to be returned to clients through context attribute evaluation. This approach uses contextual information for service discovery not for service composition. Additionally, it focuses on client-related contextual information. It does not seem to consider provider-related context which is important for Web service composition. Finally, the definition of context is limited to some attributes added to service descriptions. We adopt a more generic definition of Web service context through an ontology-based categorization of contextual information. Contextualization is proposed at the Web service deployment, composition and conciliation or matching levels in (Maamar, Benslimane, and Narendra, Maamar et al.2006). The description of contexts is assumed to occur along three categories: profile, process model, and grounding. The profile describes the arguments and capabilities of a context. The process model suggests how context collects raw data from sensors and detects changes, that need to be submitted to the Web service. Finally, the grounding defines the bindings (protocol, input/output messages,

etc.) that make context accessible to a Web service. The authors did not however mention how relevant contexts are elicited in a service matchmaking process.

CHAPTER 3

SELF-HEALING FRAMEWORK

In this chapter, we present our proposed self-healing framework to predict and recover from service faults. A service fault may be defined as the result of an unsuccessful or undesired outcome of the service operation. The fault can thus be the result of a data event such as unavailable reservation on a hotel in our running scenario, or related to execution time, when the service exceeds the expected execution time.

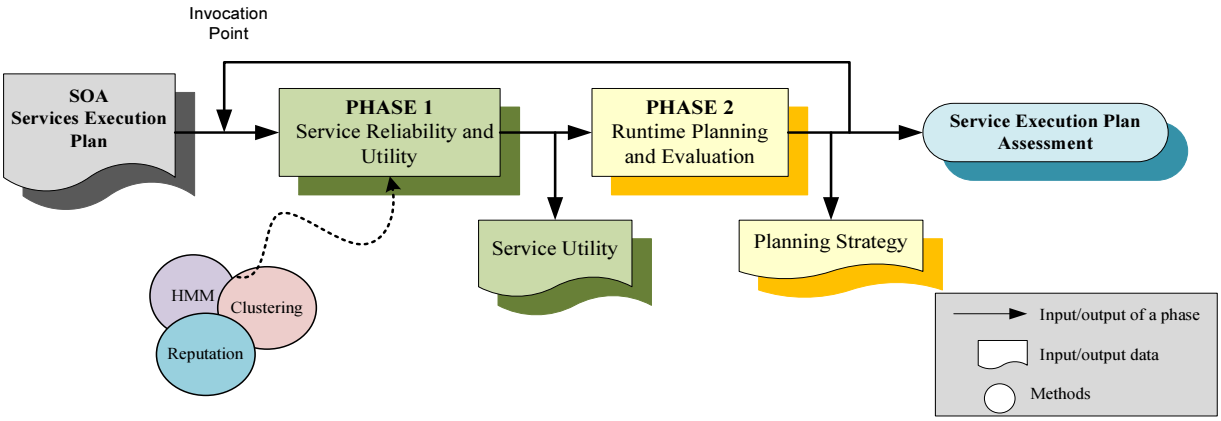


Figure 3.1: FLEX phases.

There are other types of faults such as user and network faults, but in this work we focus only on the faults that occur at the service level. To predict and recover from the failure of a service, we proposed Fault occurrence Likelihood estimation with EXception handling (FLEX). FLEX is divided into two phases: Phase I; service reliability and utility and Phase II; runtime planning and evaluation(see Figure 3.1). In Phase I, we assess the fault likelihood of the service using a combination of techniques (e.g., Hidden Markov Model, Reputation, Clustering). In Phase II, we

build a recovery plan to execute in case of fault(s) and we calculate the overall system reliability based on the fault occurrence likelihoods assessed for all the services that are part of the current composition. FLEX relies on five key activities to support dynamic management based on the changes in user requirements and QoS levels.

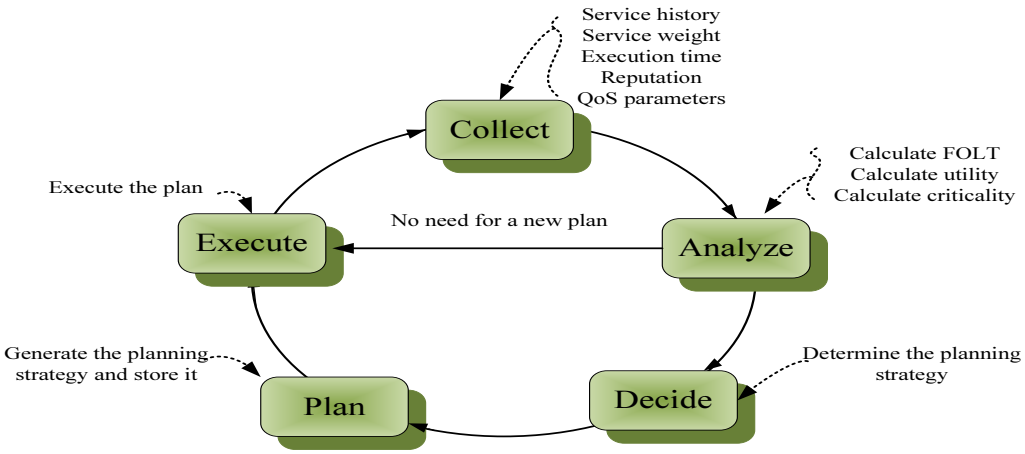


Figure 3.2: The autonomic control loop for FLEX.

Figure 3.2 shows the key activities: collect, analyze, decide, plan and execute. FLEX collects data from the system (and accompanying community) for each service such as: the service history, the service reputation, the service weight, the service execution time and other QoS parameters. The accumulated data is then analyzed by calculating the fault likelihood, the utility and the criticality of the services. In the next step, FLEX determines which one of the planning strategies is to be used (i.e., ignore, replace, etc). Then FLEX generates the new recovery plan(s) and stores for future. If the fault likelihood exceeds a pre-defined threshold, then FLEX will execute the plan immediately. These steps are described in details in the following sections.

3.1 Fault Prediction

In this section, we present the first step of FLEX which is the fault prediction. In this Phase, we calculate the fault occurrence likelihood (FOLT) for the service to assess its reliability. The notations used hereafter are listed in Table 3.1. Most of the terms in the table are self-explanatory.

Table 3.1: Definition of Symbols

Symbol	Definition
T	The total execution time.
t_0	Start time.
t_n	End time.
t_i	Time at which a new service is invoked.
k	Number of services.
$P(x)^t$	Fault occurrence likelihood for service$_x$ when invoked at time t.
λ_i	Weight of service$_i$ in relation to T.
λ'_i	Weight of service$_i$ in relation to $(T - t_i)$.
Δ_i	First-hand fault history ratio of service$_i$.
Δ'_i	Second-hand fault history ratio of service$_i$.
$f(s_i)$	The priority of service$_i$ in the composition.

Brief descriptions of other symbols follow: λ_i is the ratio of the time taken by $service_i$ (to complete its execution), to the total composition execution time. On the other hand, λ'_i is the ratio of the time taken by $service_i$ to the total time "remaining" in the composition, from the point when $service_i$ was invoked. Δ_i is the first-hand experience of an invoking service regarding a component $service_i$'s propensity to fault. For cases where the invoker has no historical knowledge of $service_i$ (i.e., the two services had no prior interaction), $\Delta_i = 0$. Similarly, Δ'_i is the second-hand experience regarding a service$_i$'s faulty behavior. This information is retrieved from other services that have invoked $service_i$ in the past. We assume that trust mechanisms (such as (Malik, Akbar, and Bouguettaya, Malik et al.2009)) are in place to retrieve and filter service feedbacks.

$f(s_i)$ is the assigned weight of a $service_i$ in the whole composition. It provides a measure for the importance of $service_i$ in relation to other component services invoked, where $\sum_{i=1}^{n} f(s_i) = 1$.

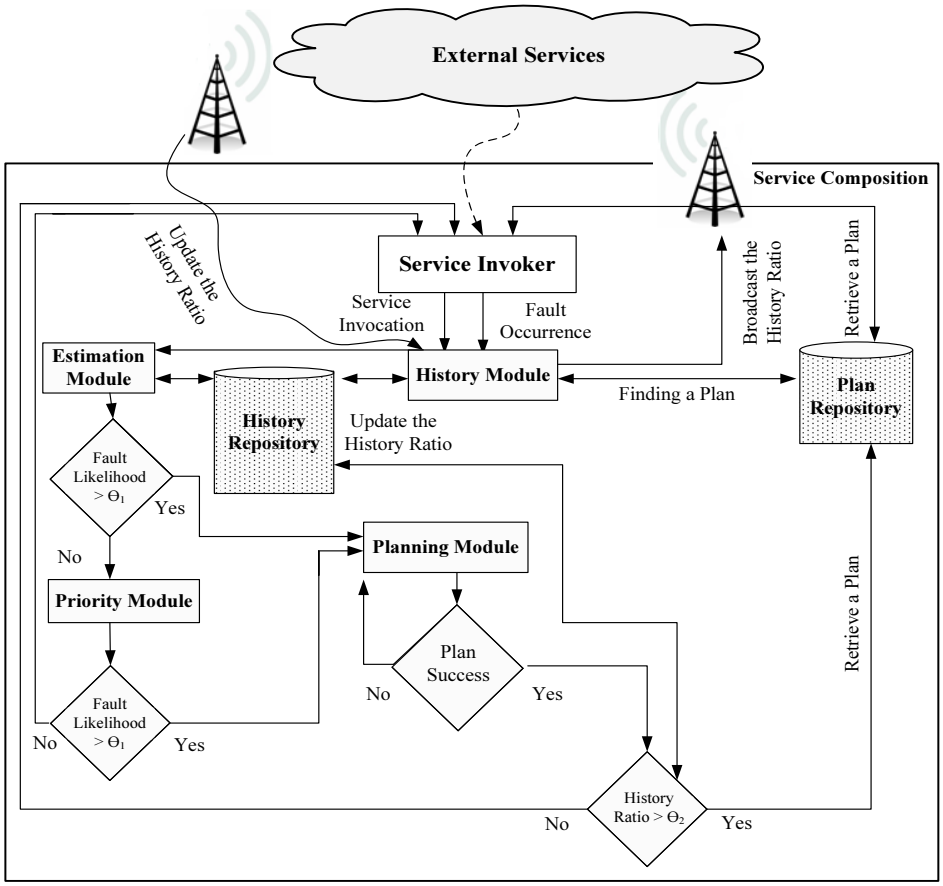


Figure 3.3: FOLT Architecture.

FOLT architecture (Figure 3.3) is composed of several modules. These are, *History Module*: This module keeps track of an individual service's propensity to fault. The information is stored in a *History Repository* that includes the service name, invocation time, reported faults (if any), and a numerical score. The *Estimation Module* calculates the fault occurrence likelihood for a service in a given context (execution history). An optional *Priority Module* is used sometimes (details to follow) to indicate the service priority assignment by the invoker in a given execution scenario.

Lastly, the *Planning Module* creates plans to recover from encountered faults, and prevent any future ones. As mentioned earlier, details of the module are not the focus of this work.

In summary, the designers store some of the plan details in a plan repository while others are generated at run time. Each plan contains specific fields such as: Plan ID, Plan Name, Plan Duration time, Plan Steps and Plan Counter. When FOLT decides to generate a plan, the system starts the dynamic generation process. The generated plan depends on the chosen invocation model. When the orchestrator invokes a service at any given time (invocation point), it calculates the fault history ratio for the invoked service. Here, we use the maximum value among the external ratio (service's second-hand experience as observed by the community) and internal ratio (first-hand experience of the orchestrator). The system then calculates the fault occurrence likelihood of the invoked service. If the likelihood is greater than a pre-defined threshold (θ_1) the system builds a fault prevention plan. Otherwise, the system re-calculates the likelihood taking into consideration the priority of the current service and compares the value again with θ_1. The purpose of this step is that non-critical services have no plans built for them, and the system can complete the execution even if a fault occurs in any of these services. The newly created plan is tested using a series of verifications. If the plan fails any of the tests, the system returns back to the planning module, and a new plan is created/checked. The process repeats for x number of times until a valid plan is found. If no plan is still found, the invoker/user is informed. Once a valid plan is created, it is stored in the repository. Then, If the likelihood is greater than another pre-defined threshold (θ_2) the system can execute this fault prevention plan.

Phase 1 is divided into multiple steps: calculating the service's weight (λ), calculating the

time weight (λ'), calculating the internal history value (Δ_i) using a Hidden Markov Model, and calculating the external history value (Δ'_i) using clustering and reputation. The likelihood of a fault occurring at time t is defined by studying the relationship between the service's importance, time it takes to execute, and its past performance in the composition. Thus, each invocation model will have a different fault likelihood value. As mentioned earlier, λ is the ratio of the time that is needed to complete the service execution, divided by the total time of completing the execution of the whole system. Similar to the approach used in (Meulenhoff, Ostendorf, Živković, Meeuwissen, and Gijsen, Meulenhoff et al.2009), we use this value of λ as one of the basic constructs in FOLT to measure the (relative) weight of the invoked service to the rest of system time. The basic premise is that the likelihood of a fault occurrence for a long running service will be more than a service with very short execution time. Determining the service execution time could be accomplished in two ways. If the system does not know the execution time for a service, then the service's advertised execution time is used. On the other hand, after attaining experience with the service (prior invocations), the service execution time could be recorded and stored in the repository. Then:

$$\lambda_i = \frac{T(s_i)}{T} \tag{3.1}$$

$$\lambda'_i = \frac{T(s_i)}{T - t_i} \tag{3.2}$$

where λ_i is as described above, $T(s_i)$ is the total execution time of $service_i$, while t_i is the invocation time of $service_i$ (i.e., when the service was invoked).

A service's past behavior is assessed according to first-hand experience of the invoking service and second-hand experiences of other services obtained in the form of ratings via the community.

These *experiences* are evaluated as a ratio of the number of times the service failed, divided by the total number of times the service was invoked. To assess the First-hand Experience, we use a Hidden Markov Model (HMM). The HMM provides the probability that the service will fail in the next invocation, based on the previous behavior of the service within the system. HMMs have proven useful in numerous research areas for modeling dynamic systems (Malik, Akbar, and Bouguettaya, Malik et al.2009). An HMM is a process with a set of states, set of hidden behavior and a transition matrix. In our architecture, all services stay in one of the two states: Healthy or Faulty (Figure 3.4).

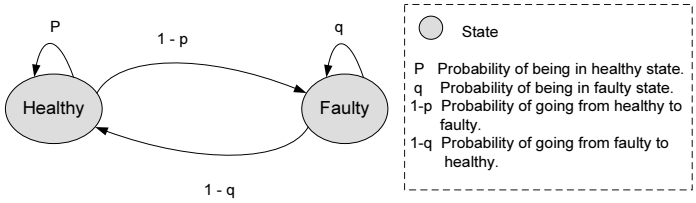


Figure 3.4: Finite state machine for an HMM of the service.

Each time the composition orchestrator invokes a service, it records the state of that service (Faulty or Healthy) along with the time of the invocation. Let the vector V = the service behavior profile, then to asses the probability that $Service_i$ will be in the Faulty state in the next time instance:

$$P(Faulty|V) = P(Faulty|Healthy) + P(Faulty|Faulty) \tag{3.3}$$

FOLT also uses other services' experiences with $Service_i$ to assess its reliability. Services are divided into clusters based on their similarity (such as in (Abramowicz, Haniewicz, Kaczmarek, and Zyskowski, Abramowicz et al.2007)). These group of services are consulted for the reputation

of $Service_i$. We assume that other services are willing to share their reputation ratings, which are assimilated using our previous work in (Malik, Akbar, and Bouguettaya, Malik et al.2009). A combination of service time weights and service history ratios (using HMM, and reputation) is used to assess the fault occurrence likelihood:

$$P(s_i)^t = 1 - (\lambda_i')^{\frac{\lambda_i}{1-max(\Delta_i,\Delta_i')}} \tag{3.4}$$

Note that the fault history is assessed according to $max(\Delta_A, \Delta_A')$. Then, the likelihood of a service executing without any fault is $1 - max(\Delta_A, \Delta_A')$. We use this value in relation to the total execution times (remaining given by λ', and overall given by λ) to assess the likelihood of a service executing *without* a fault. To get the likelihood of the service's fault occurrence we subtract this value from 1 in Equation 3.4. In cases where we need to incorporate a service's priority weight, Equation 3.4 becomes:

$$P(s_i)^t = 1 - (\lambda_i')^{\frac{\lambda_i f(s_i)}{1-max(\Delta_i,\Delta_i')}} \tag{3.5}$$

Appendix 1 shows the mathematical proof for Equation 3.5. We observe that with increased service priority, fault likelihood also increases. Based on the fault likelihood, FOLT decides when to build a recovery plan. Services with a high priority are usually critical, and a fault in any of those services may harm the overall QoS. Thus, fault likelihood and service priority are directly proportional in FOLT.

Using Equation 3.5 as the basis, we define fault likelihood estimation for each invocation model. For instance, the likelihood of fault(s) in a sequential invocation (P_{seq}) is dependent on the successor service(s) (Cardoso, Miller, Sheth, and Arnold, Cardoso et al.2002). Since FOLT

uses invocation points, only a single service can be invoked per time instance/invocation point. Hence the equation stays the same. Let A be the successor service, then

$$P_{seq} = P(s_A)^t = 1 - (\lambda'_A)^{\frac{\lambda_A f(s_A)}{1-max(\Delta_A, \Delta'_A)}} \tag{3.6}$$

In a parallel invocation, fault estimation at the invocation point translates to the fault occurring in either of the invoked services. Since all services are independent, we need to add their fault likelihoods. Moreover, due to the likelihood of simultaneous faults occurring in the respective services, we have

$$P_{par} = \bigcup_{i=1}^{h} P_i = \Sigma_{i=1}^{h} P_i - \Pi_{i=1}^{h} P_i$$

$$P_{par} = \Sigma_{i=1}^{h}(1 - (\lambda'_i)^{\frac{\lambda_i f(s_i)}{1-max(\Delta_i, \Delta'_i)}}) - \Pi_{i=1}^{h}(1 - (\lambda'_i)^{\frac{\lambda_i f(s_i)}{1-max(\Delta_i, \Delta'_i)}}) \tag{3.7}$$

where h is the number of services invoked in parallel.

In probabilistic invocation (P_{pro}), fault likelihood depends on the probability of selecting the service (Q). Then, if we have k services:

$$P_{pro} = \bigcap_{i=1}^{k} P_i = \Pi_{i=1}^{k} Q_i \times P_i \tag{3.8}$$

Similarly, the fault likelihood of a circular invocation is:

$$P_{cir} = \Pi_{i=1}^{n} P_S \tag{3.9}$$

3.2 Run-time Planning

In this section, we present the second step of FLEX which is the run-time planning. Services involved in an SOA often do not operate under a single processing environment and need to communicate using different protocols over a network. Under such conditions, designing a fault management system that is both efficient and extensible is a challenging task. In essence, SOAs are distributed systems consisting of diverse and discrete software services that work together to perform the required tasks. Reliability of an SOA is thus directly related to the component services' behavior, and sub-optimal performance of any of the components may degrade the SOA's overall quality. The problem is exacerbated due to security, privacy, trust, etc. concerns, since the component services may not share information about their executions. This lack of information translates into traditional fault management tools and techniques not being fully equipped to monitor, analyze, and resolve faults in SOAs.

In Phase I (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2011), we defined a fault management approach (Fault Occurrence Likelihood esTimation: FOLT) for SOAs. We assume that component services do not share their execution details with the invoking service (defined as an orchestrator). The orchestrator only has information regarding the services' invocation times and some other *observable* quality of service (QoS) attributes. Once faults are identified (i.e., likely to occur in the future), recovery plans need to be created. However, fault recovery plan generation is challenging due to the lack of capabilities in current systems to adapt themselves at run time to cope with dynamic changes in user requirements and the running levels of QoS attributes (Nascimento, Rubira, and Lee, Nascimento et al.2011) (Mulo, Zdun, and Dustdar, Mulo

et al.2010). In order to support such dynamic changes, we propose FLEX (FoLt with EXception handling) a fault-tolerant mechanism which combines our planning strategies based on FOLT calculations (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2011) and incorporates the exception handling constructs of BPEL. Our planning module captures both the functional and non-functional features of Web services. Functionality is specified by the operations offered by a Web service, while the non-functional part comprises the QoS properties of a Web service.

Specifically, we propose a novel technique to dynamically evaluate the performance of Web services based on their previous history (in terms of QoS), and user requirements. The likelihood of fault occurrence is then used to create (multiple) recovery plans. The 'best' recovery plan is then chosen to be either executed immediately (if fault likelihood is above a pre-defined threshold), or saved for a later execution (i.e., to be executed when the fault occurs).

3.2.1 Utility and Reliability Calculation

The reliability of a service is determined as the percentage where the service is providing fault free service or in the worst case the minimum number of faults. This value is used in a utility function to determine whether we need to create a fault recovery. A utility function R(λ'_i) defined for $0 < \lambda'_i < 1$ has the property of non-satiation (i.e., the first derivation $R'(\lambda'_i) > 0$, and the second derivation $R''(\lambda'_i) < 0$). Thus:

$$R(S_A) = (\lambda'_A)^{\frac{\lambda_A f(s_A)}{1-max(\Delta_A, \Delta'_A)}} \tag{3.10}$$

We first find the first derivative for λ_i' based on the above equation as:

$$R'(S_A) = \frac{dR}{d\lambda_A'} = \frac{\lambda_A f(s_A)}{1 - max(\Delta_A, \Delta_A')}(\lambda_A')^{\frac{\lambda_A f(s_A)}{1-max(\Delta_A,\Delta_A')}-1} \tag{3.11}$$

By analyzing Equation 3.11 we can see that this will generate a positive value since:

$$0 < \lambda_A' < 1 \tag{3.12}$$

$$0 < f(x) < 1 \tag{3.13}$$

$$max(\Delta_A, \Delta_A') < 1, \quad 0 < 1 - max(\Delta_A, \Delta_A') < 1 \tag{3.14}$$

then

$$\frac{\lambda_A f(s_A)}{1 - max(\Delta_A, \Delta_A')}(\lambda_A')^{\frac{\lambda_A f(s_A)}{1-max(\Delta_A,\Delta_A')}-1} > 0 \tag{3.15}$$

i.e.,

$$\frac{dU}{d\lambda_A'} > 0 \tag{3.16}$$

Similarly,

$$R''(S_A) = \frac{d^2R}{d{\lambda_A'}^2} \tag{3.17}$$

$$R''(S_A) = (\frac{\lambda_A f(s_A)}{1 - max(\Delta_A, \Delta'_A)})^2 (\frac{\lambda_A f(s_A)}{1 - max(\Delta_A, \Delta'_A)} - 1)(\lambda'_A)^{\frac{\lambda_A f(s_A)}{1 - max(\Delta_A, \Delta'_A)} - 2} \tag{3.18}$$

By analyzing Equation 3.18 we can see that thus will generate a negative value since:

$$\frac{\lambda_A f(s_A)}{1 - max(\Delta_A, \Delta'_A)} < 1 \tag{3.19}$$

subtracting 1 from each side, gives

$$\frac{\lambda_A f(s_A)}{1 - max(\Delta_A, \Delta'_A)} - 1 < 0 \tag{3.20}$$

The minimum expected value for λ_A, $f(s_A)$, and $(1 - max(\Delta_A, \Delta'_A))$ is equal to 0.01 then the minimum value of this equation is 0.01, thus, (0.01 - 1 = -0.99). However, the maximum expected value for the previous parameters is 0.99 then (0.99 - 1 = -0.01). So, $(\frac{\lambda_A f(s_A)}{1 - max(\Delta_A, \Delta'_A)} - 1)$ will be always less than 1 which makes the whole equation (Equation 3.18) negative.

$$\frac{d^2 U}{d{\lambda'_A}^2} < 0 \tag{3.21}$$

Form the Equations 3.11 and 3.18 we can see that our equation is a valid objective utility function (as per the definition). As mentioned earlier, the utility value is used to make a decision regarding plan generation/execution based on the following:

$$R = \begin{cases} Rmax & \text{if } 0.7 < R \leq 1; \\ Raverage & \text{if } 0.4 < R \leq 0.7; \\ Rmin & \text{if } 0.2 < R \leq 0.4; \\ Rrisk & \text{if } 0 \leq R \leq 0.2; \end{cases}$$

If the utility is *Rmax* that means the service is expected to execute correctly without possibility of a fault. In this case, FLEX invokes the service without any concern and no recovery plan is generated. However, if the utility is *Raverage* that means the service has a possibility of failure but this possibility is low. In such a case FLEX will create a recovery plan and store it in the plan repository and when the fault occurs, it retrieve and execute the created plan. When the value of utility is *Rmin* that means the service has a high possibility of failure, and FLEX should create and execute the recovery plan instead of invoking the service. Finally, if the utility is *Rrisk* then the system should take an immediate action by replacing the service by a similar one.

3.2.2 Dynamic Recovery Plan Generation

Planning can be defined as "a kind of problem solving, where an agent uses its beliefs about available actions and their consequences, in order to identify a solution over an abstract set of possible plans" (Jensen, Jensen2004). In any composite system two main types of services may exist: critical services (CS) and non-critical services (NS). The degree of criticality is thus based on the tasks undertaken by these services, where each task contains one or more operations. In FLEX, to determine the criticality degree of a service we use: (i) user priority and operation weight calculation (UOW), and (ii) the critical path method (CPA). In UOW, the main factors are:

the operation priority value that is provided by the consumer ($Wuser_i$) and operation criticality (defined as the ratio of the operation's execution time to the total execution time of the service).

$$OpCritical(op_i, service_j) = \frac{W_{Op_{ij}} \times \sum_{k=1}^{n} R_k}{|W| + |D|} \tag{3.22}$$

Where ($W_{Op_{ij}}$) is the weight of the operation i in service j which is calculated by dividing the execution time of the operation over the total service time, (R_k) is the reputation of the service that is provided by n different composition, W is the number of component services in the composition, and D is the number of Web services that depend on Web service j. After determining the criticality of the operation we calculate the criticality of the task, and the criticality of the overall service ($ServiceSignificance$).

$$TaskCriticality(task_k) = \frac{\sum_{i=1}^{z} Wuser_i \times OpCritical(op_i, service_j)}{\sum_{i=1}^{z} Wuser_i} \tag{3.23}$$

where z is the number of operations in task$_k$, and $Wuser_i$ is the user's weight for op$_i$.

$$ServiceSignificance = \sum_{i=1}^{m} TaskCriticality(task_i) \tag{3.24}$$

where m is the number of tasks in the service$_j$. FLEX uses the $ServiceSignificance$ value to create a binary decision for UOW. If $ServiceSignificance > \beta$ (where β is a predefined threshold) then service criticality value for UOW is one, otherwise it is zero. The binary UOW value is used in conjunction with the CPA value. The discussion follows.

In CPA, we specify the composition in the system as a statechart (Zeng, Benatallah, Dumas,

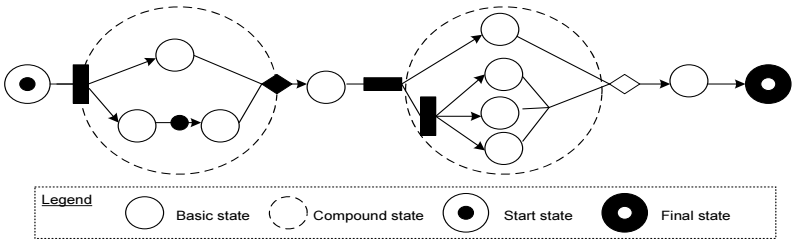


Figure 3.5: Statechart of travel scenario (SURETY).

Kalagnanam, and Sheng, Zeng et al.2003). The choice of statechart is for two reasons: it has well-defined semantics and it offers the basic flow invocations that exist in any composition such as sequential, parallel and circular. In our approach, the states are the Web service(s), and the transitions among the states are the links to the next invocation point. States can be basic or compound. Basic states are labeled to one Web service. However, compound states contain more than one Web service. An example for a compound state is a set of Web services (S1,S2, and S3). As a simplified statechart Figure 3.5 shows the statechart for our running scenario which contains nine basic states and two compound states.

Definition 1:(*Execution path*) An execution path of a statechart is a sequence of basic states [State$_1$, State$_2$,..,State$_n$], where State$_1$ is the initial state, State$_n$ is the final state, and State$_i$ $\in$ (State$_1$,..,State$_n$). State$_i$ is a direct successor of one of the states in [State$_1$,..,State$_{i-1}$] and it is not a direct successor of the states in [State$_{i+1}$,..,State$_n$].□

From Definition 1, we see that the statechart has a finite number of paths. In addition, if the statechart has probabilistic invocation, it has multiple execution paths, where each one represents a sequence of services.

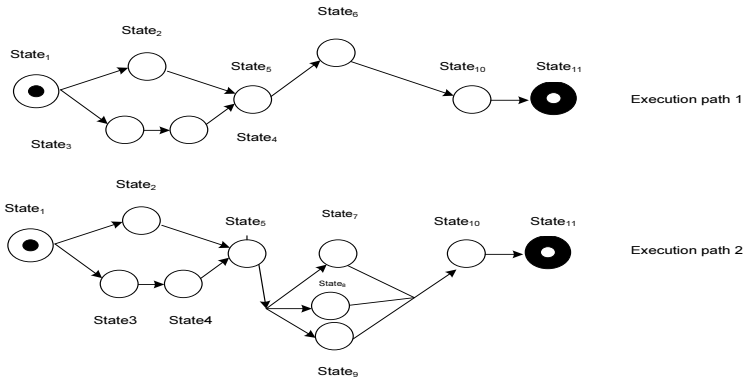


Figure 3.6: Representation of different execution paths.

Figure 3.6 gives an example of two execution paths for the statechart in Figure 3.5. Since there is a probabilistic branch after $state_6$, there are two paths: path 1 and path 2. In path 1 the execution goes from $state_5$ to $state_6$ then $state_{10}$, while in path 2 the execution goes from $state_5$ to all the other states ($state_7$, $state_8$ and $state_9$) then $state_{10}$. We use the critical path algorithm (CPA) for determining the critical services in the system. The critical path of a system is a path from the start state to the final state which has the highest/lowest total sum of weights in the consumers view of point. In our running example, we assume that the consumer's main concern is the execution time, and each service's execution time is given in Figure 3.7. After calculating the total execution time for each path, we found that the minimum total execution time is 89 ms (execution path 1). The critical path is thus ($state_3$, $state_4$, $state_5$, $state_6$, and $state_{10}$). Every service located in the critical path is critical service. After determining the critical services using CPA and UOW we are use a conjunction (XOR) to determine the final decision for criticality. The service criticality is equal to zero if (CPA = 0 AND UOW = 0).

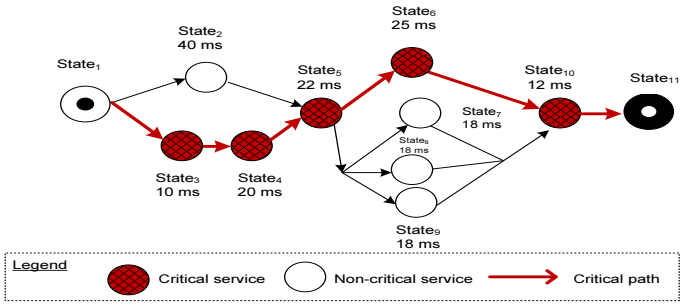


Figure 3.7: Critical path and critical services.

3.2.3 Incorporating WS-BPEL

Business Process Execution Language (BPEL) is a commonly accepted standard for defining business processes with composition of services in SOA (Andrews, Curbera, Dholakia, Goland, Klein, Leymann, Liu, Roller, Smith, Thatte, Trickovic, and Weerawarana, Andrews et al.2003). Standard BPEL variables are a snapshot of data returned by a service and thus present a duplicate version of remote data at some particular time. However, in certain cases we want to ensure that the process always uses the latest version of important business data as other applications may change the data in the data source during the BPEL process execution.

This problem arises particularly in case of long-running BPEL processes, which can take a few days, weeks or even months to complete. WS-BPEL has a few built-in exception handling strategies, that work similar to the try-throw-catch mechanism in programming languages such as JAVA. Such exception handling mechanisms work after a service is invoked and a fault occurs. However, the FLEX planning module also enables the creation of a recovery plan before service innovation (and execute this alternate in special cases). Figure 3.8 shows the general architecture of FLEX. As mentioned earlier, FLEX works in two different ways (i.e., before or after the occurrence of the

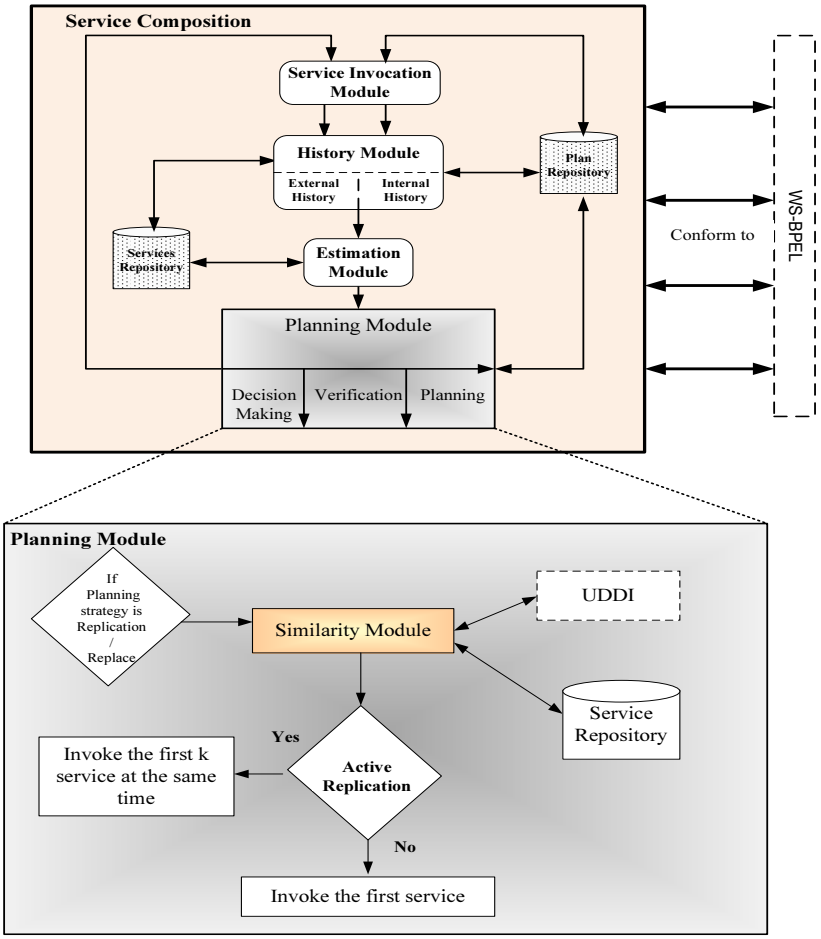


Figure 3.8: Overview of FLEX.

fault): *pre-recovery* and *post-recovery*. In the *pre-recovery*, FLEX works alone without WS-BPEL, but in the *post-recovery*, FLEX fully exploits WS-BPEL's built-in exception handling constructs. The *pre-recovery* actions are predicting the fault, building the recovery plan, and executing the recovery plan (it needed). The primary *post-recovery* action is using WS-BPEL to handle the fault, and if BPEL's built in strategies can not recover from the fault, then as a secondary action, FLEX reinitiates the planning process.

In general, before invoking any service the Service Invocation Module sends the service's infor-

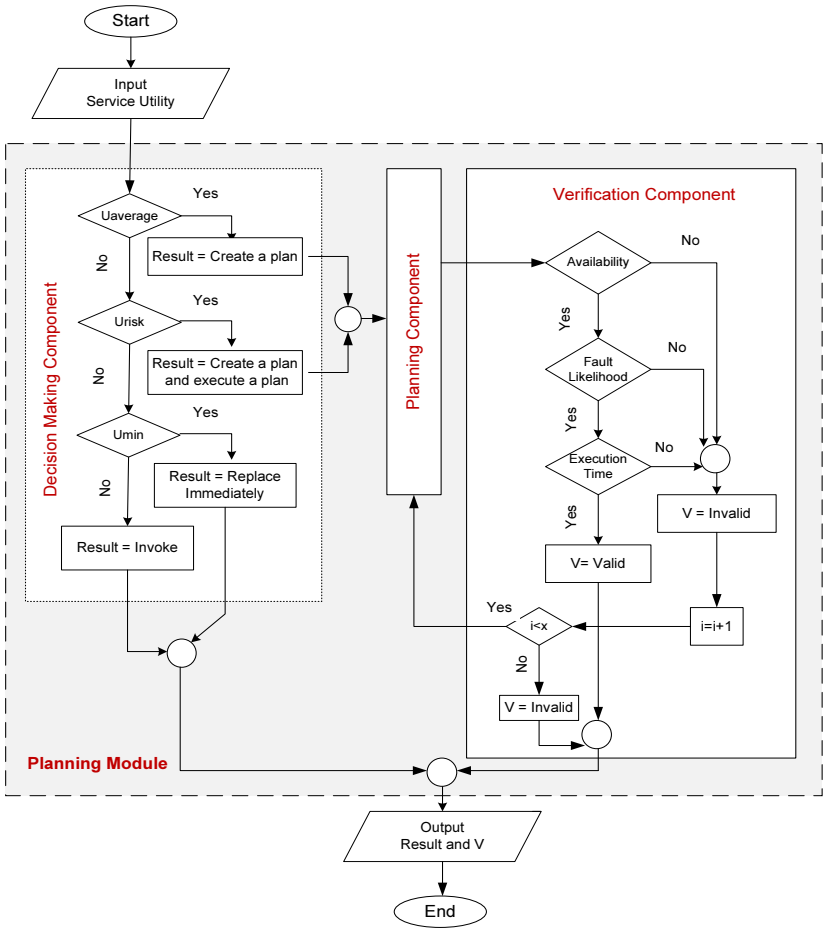


Figure 3.9: Planning module processes.

mation to the History Module which is responsible for calculating the fault likelihood (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2011). Based on this value, the Estimation Module determines if the system needs *pre-recovery*, or it can continue the invocation process. If the decision is *pre-recover* then the Planning Module decides the best plan through the Decision Making Module, i.e., generates the recovery plan, verifies it, stores it in the Service Repository, and execute it (if needed). When a fault occurs at runtime, the system first employs the WS-BPEL exception handling strategies to repair it. If the fault is fixed, the composite service continues its execution.

Otherwise, the system returns back to the Planning Module which generates an optimal recovery plan and executes it. The Planning Module (PM) consists of three components: Decision Making Component (DMC), Validation Component (VC) and Planning Component (PC). Each one of the components has it is own responsibilities. Starting from DMC, the input of this component is the service utility which could be (as mentioned earlier) one of four utility values such as: Rmax, Raverage, Rrisk and Rmin. Based on the utility of the service, DMC decides whether to create a plan or not. A new/generated plan is then validated (using VC). The decision of VC is based on the availability of the plan (i.e, whether the services in the new plan are available), the fault likelihood of the plan, and the execution time of the new plan compared to the current execution time. If VC determines that the plan is valid, the likelihood of fault is low and the execution time is acceptable, then the plan is stored in the plan repository. On the other hand, if the plan fails in any one of the tests, the system returns back to PC, and a new plan is created/validated. The process repeats for x number of times until a valid plan is found. If no plan is still found, the invoker/user is informed, and if required FLEX invokes the current service plan with the expected risks (see Figure 3.9). The six exception handling strategies used in FLEX: Ignore, Parallel-Retry, Retry-Until, Replace, Active Replication and Passive Replication. Details follow.

Ignore: Just as its name suggests, this strategy does not take any direct action to handle a fault, other than ignoring the execution of the faulty service (i.e., the system may ignore the current service and continue the execution without wasting time in fixing the fault). This strategy is used in situation where the service ignorance will not affect the main goal. For instance, in our scenario

(Figure 1.3), if there is a fault in S_3 (e.g., no response after specific time) FLEX may ignore S_3 and skip to service S_4 (Figure 3.10).

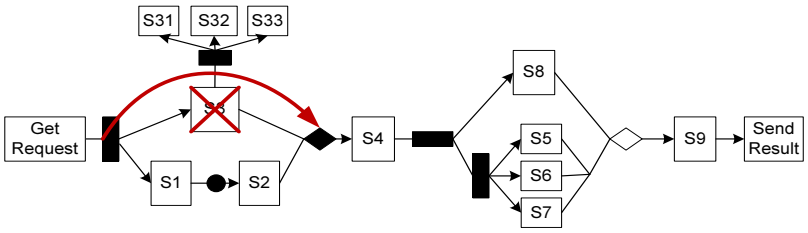


Figure 3.10: Ignore Strategy.

Precondition: `Fault or fault predicted , and Service`$_x$ `is not critical`

1. `Disable Service`$_x$, i.e., disable the invocation of this service.

2. `Return back` to the last step in the execution plan before invoking the faulty service.

3. `Create a bypass link` to the immediate next service in the system.

4. `Resume the execution` of the 'new' composition.

5. `Report the state` (e.g., a fault report) to the system.

Retry: In this strategy, if a fault occurs in a service, the system repeats the same service once. The repetition of the service in Retry may be carried out without altering any conditions or changing some of the conditions (e.g., input variables, etc.). Figure 3.11 shows an example of retry where the *Hotel booking (S2)* service was repeated because a fault occurs in this service (e.g., change of flight date). In this case, the strategy deletes all previous results and restarts from the beginning.

Precondition: `Fault or fault predicted, and Service`$_x$ `is available`

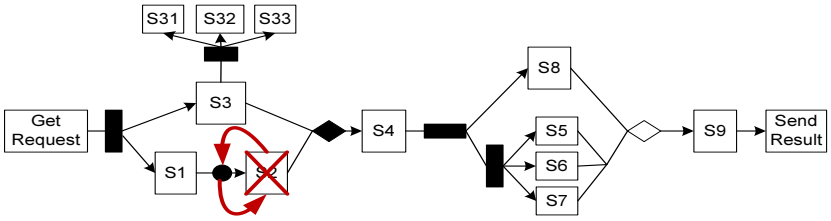


Figure 3.11: Retry Strategy.

1. `Delete results` from the first time invocation for Service$_x$.

2. `Return back` to the first step in the execution plan of invoking the faulty service.

3. `Change any variables` if required.

4. `Resume the execution`.

5. `Report the state` (e.g., a fault report) to the system.

Retry-Until: In the retry-until strategy, if a fault occurs in a service, the system repeats the same service multiple times. The repetition of the service is with a condition that the execution time should not exceed ρ, where ρ is the maximum time the system allows for this service's execution. For instance, if ρ = 40 ms, and the *Hotel booking* service time is equal to 15 ms, then this service will be repeated for two times at maximum.

Precondition: `Fault or fault predicted , Service`$_x$ `is available,` ρ

1. `Delete results` from the first time invocation for Service$_x$.

2. `Return back` to the first step in the execution plan of invoking the faulty service.

3. `Check condition`: Check the maximum acceptable time

   - `Resume the execution`.

4. `Report the state` (e.g., a fault report) to the system.

Replace: Under the replace strategy, the system should find another similar service and replace the current one. The new service should have the same functional and non-functional parameters. This is especially interesting in the context of Web services since each Web service is represented by a category. In our scenario (Figure 3.12), if the *Flight reservation (S1)* service fails then the system looks for similar services and replace it with the new one (i.e., the same category under UDDI).

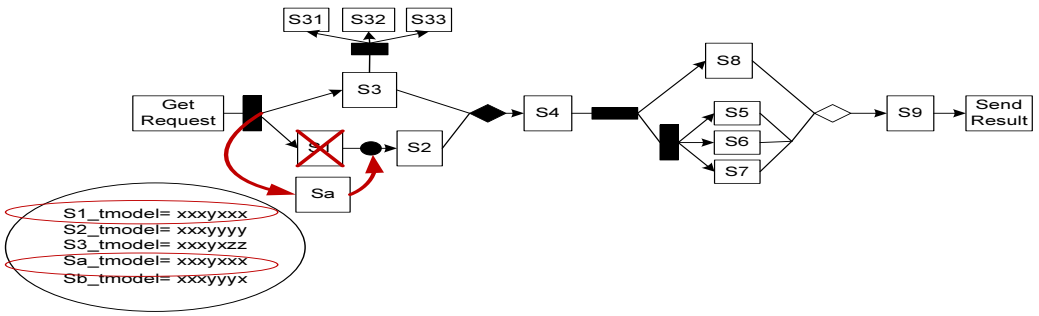


Figure 3.12: Replace Strategy.

In FLEX, we have two types of replacement: overlapping and full replacement. In overlapping, we keep the operations that perform well and replace the operations that have faults, while in full replacement, all service operations are replaced (see Figure 3.13). Otherwise, if the system cannot find that kind of operation, it moves to full replacement of such as service$_a$ (Nepal, Sherchan, Hunklinger, and Bouguettaya, Nepal et al.2010) (Menascé and Dubey, Menascé and

Dubey2007) (Bergmann, Richter, Schmitt, Stahl, and Vollrath, Bergmann et al.2001) (Li, Xu, Wu, and Zhu, Li et al.2012) (Yao, Lu, Fu, and Ji, Yao et al.2010) (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2012).

Precondition: `Fault or fault predicted, and Service`$_y$ `is available`

1. `Delete results` from the first time invocation for Service$_x$.
2. `Return back` to the first step in the execution plan of invoking the faulty service.
3. `Similar service`. i.e., find a service Service$_y$ which is similar to Service$_x$.
4. `Create link` to the new alternative service (Service$_y$).
5. `Resume the execution`.
6. `Report the state` (e.g., a fault report) to the system.

Replicate: In replicate, the system invokes multiple equivalent services at the same time (equivalent means that they provide the same functional and (similar) non-functional parameters). Thus, if a service fails, then there is another service that can fulfill the desired task. In our scenario, when the strategy is replicate for *Transportation service (S4)* then the system invokes the similar services (S_b, S_c and S_d). (see Figure 3.14).

Precondition: `Fault or fault predicted , and similar services are available`

1. `Delete results` from the first time invocation for Service$_x$.
2. `Return back` to the first step in the execution plan of invoking the faulty service.
3. `Keep the link` to Service$_x$.

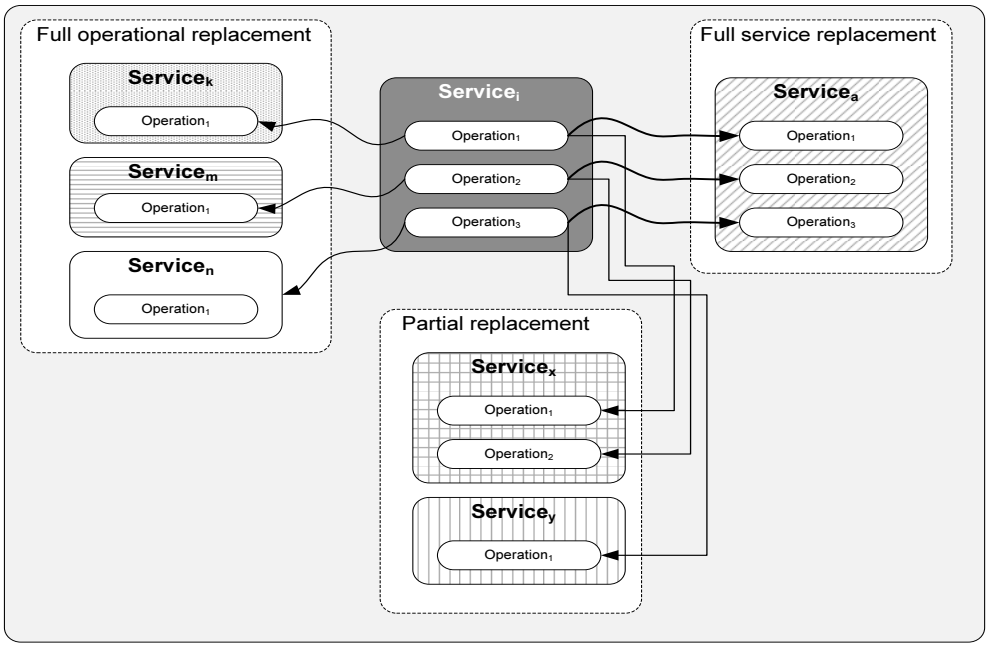


Figure 3.13: Different scenarios of replace strategy.

4. `Similar service`. i.e., find services that is similar to Service$_x$.

5. `Create link` to the new alternative service (Service$_y$).

6. `Resume the execution`.

7. `Report the state` (e.g., a fault report) to the system.

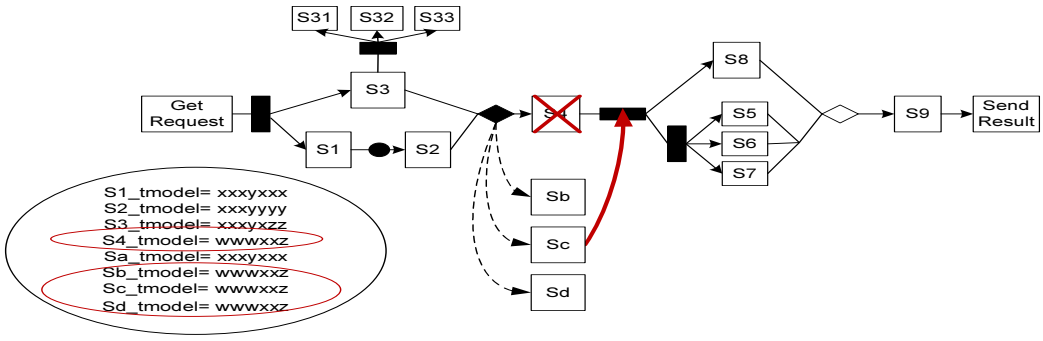


Figure 3.14: Replicate Strategy.

In replication we have two types: active and passive replication. In the active replication, the system invokes all replicas at the same time and takes the service that returns the first response. Some fault tolerance techniques employee active replication such as Fewer (Santos, Lung, and Montez, Santos et al.2005), Thema (Merideth, Iyengar, Mikalsen, Rouvellou, and Narasimhan, Merideth et al.2005) and WS-Replication (Salas, Perez-Sorrosal, Pati and Jiménez-Peris, Salas et al.2006). In passive replication, the system invokes the primary replica to perform the job and invokes backup replicas only if the primary replica fails. Some of the fault tolerance techniques employee passive replication such as FT-SOAP (Fang, Liang, Lin, and Lin, Fang et al.2007) and FT-CORBA (Majzik and Huszerl, Majzik and Huszerl2002). Table 3.2 summarizes the previous recovery strategies.

Table 3.2: FLEX planning strategies

Recovery plan	Brief description
Ignore (Service$_x$)	Ignores the current service and skips to the immediate next service(s).
Retry (Service$_x$)	Repeats the execution of the current service once.
Retry-until (Service$_x$, ρ)	Repeats the execution of the current service until the service execution time exceeds ρ.
Replace(Service$_x$, Service$_y$)	Executes an alternative of service$_x$ (i.e., replace by service$_y$).
Passive-replication (Service$_x$, Service$_y$)	Allows the execution of service$_x$ and service$_y$ at the same time.
Passive-replication (Service$_x$, ServicesSet$_{ss}$)	Allows the execution of service$_x$ and all the services in ServiceSet at the same time.

FLEX creates dynamic plans (when the system needs them at run time) based on the utility and criticality of the services.

- **Utility (R):** This is the utility of the service which has been calculated in the previous part:

Rmax, Raverage, Rmin and Rrisk.

- **Criticality (C):** This is a binary value assessed from the two different ways (UOW and CPA) by calculating the criticality degree.

- **Ucost:** This is a binary value that the user provides to the system. If the user does not accept any extra cost (i.e., the cost of invoking another services), Ucost will be zero. When Ucost equals to one that means the user does not mind the extra cost.

- **Utime:** This is a binary value that the user provides to the system. If the user does not accept any extra time, Utime will be zero. When Utime is equal to one that means the user does not mind an increased execution time.

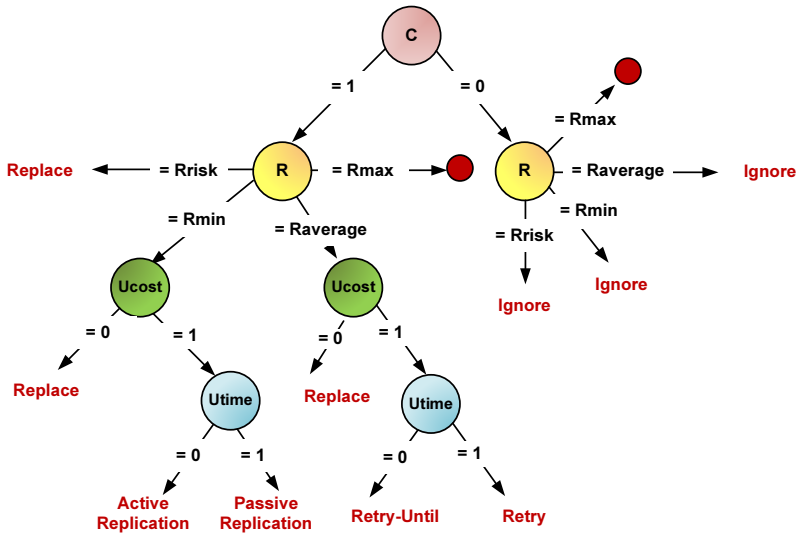


Figure 3.15: Selection Tree for Planning Strategies.

Based on the above notations, we build a selection tree. Figure 3.15 shows our selection tree which consists of four levels. In the first level, the system checks the service criticality (C), if the

criticality is equal to zero then it goes to the second level to check the utility (R). If the utility is Rmax then there is no need to build any plan, but if the utility is Raverage, Rmin or Rrisk then the system will ignore the service in case of fault because it is not a critical service. On the other hand, when the criticality (C) is equal to one that means the current service is critical. Thus, if the service utility is Rrisk then the recovery plan is replace the service immediately before the current invocation, but if the utility is Rmin then the system moves to the third level to check Ucost. If Ucost is equal to zero then the planning strategy is replacing the service, but if Ucost is equal to one then the system moves to the fourth level which checks Utime. In the case that Utime is equal to zero, then the planning strategy is active replication because the user accepts extra cost but he does not accept extra time. If Utime is equal to one, then the choice is passive replication because the user accepts both extra cost and time. When the Utility (R) is Raverage the system checks Ucost, if it is equal to zero then the planning strategy is replace, but if it is equal to one then the system checks Utime. If Utime is equal to zero, then the plan is Retry-Until, but if Utime is equal to one then the plan is Retry.

CHAPTER 4

SEMANTIC SIMILARITY AND RANKING

In this chapter, we present our proposed Semantic Similarity and Ranking framework. Due to the competitive and fast growing nature of today's business climate, most organizations are automating their business processes for service and operation delivery. In this respect, service-oriented computing (SOC) has become a main trend in software engineering that exploits Web services as fundamental elements for developing on-demand applications. Web services are self-described, self-contained and platform-independent computational elements that can be published, discovered, and composed using standard protocols, to build applications across various platforms and organizations in a dynamic manner. With the increasing agreement on the functional aspects of Web services, such as using WSDL for service description, SOAP for communication and WS-BPEL for composing Web services etc., the research interest is shifting towards the non-functional aspects of Web services (Papazoglou, Pohl, Parkin, and Metzger, Papazoglou et al.2010).

Developers can now add descriptions (using standards such as OWL-S) to their Web services to define and advertise the non-functional aspects of services (including input, output, pre-condition, post-condition and functions), thereby facilitating automated discovery, invocation and inter-operation. However, the first step in this process is to 'resolve' the consumer request against prospective Web services, so that the most appropriate component could be selected (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2011). The expected availability of a large number of highly specialized component services, means that it would be increasingly challenging to find the most suitable service(s) in a reasonable amount of time (Nepal, Sherchan, Hunklinger, and

Bouguettaya, Nepal et al.2010). Moreover, some Web services may not be able to satisfy consumer requests individually, and hence need to be integrated with other Web services to provide the desired functionality. This adds to the complexity of an already challenging problem (Bouguettaya, Krüger, and Margaria, Bouguettaya et al.2008).

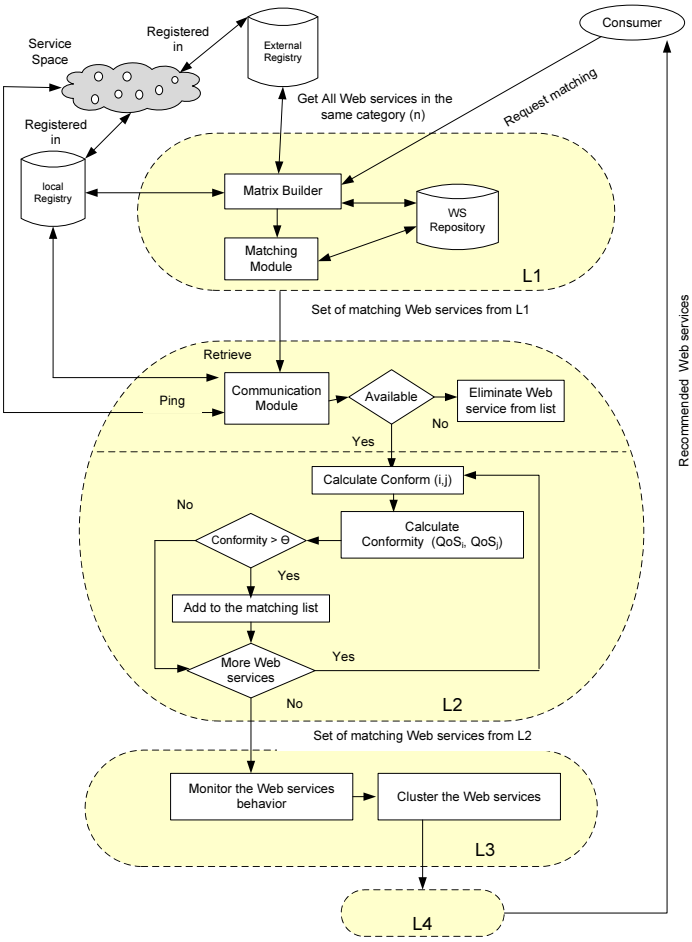


Figure 4.1: Matching levels.

In this chapter, we present our novel approach (defined S^{2}R: Semantics-based Similarity and Ranking) for Web service selection (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2012). S^{2}R is divided into four levels as shown in Figure 4.1. In the first level (L1), we filter the available Web services under a specific category based on their functional properties such as input, output

and operations. In the second level (L2), we further reduce the service search space based on non-functional properties, such as Quality of Service (QoS) parameters (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2011). In the third level (L3), we further reduce the service search space based on their behavioral properties, and we rank the services based on their utility value (in the fourth level (L4)). The utility value is calculated using a utility function which allows stakeholders to ascribe a value to the usefulness of the overall system as a function of several QoS attributes such as response time, availability, cost, reliability, etc. according to their preferences (Nepal, Sherchan, Hunklinger, and Bouguettaya, Nepal et al.2010) (Li, Xu, Wu, and Zhu, Li et al.2012) (Yao, Lu, Fu, and Ji, Yao et al.2010) (Nejati, Sabetzadeh, Chechik, Easterbrook, and Zave, Nejati et al.2007).

The utility value is calculated using a utility function which allows stakeholders to ascribe a value to the usefulness of the overall system as a function of several QoS attributes such as response time, availability, cost, reliability, etc. according to their preferences (Nepal, Sherchan, Hunklinger, and Bouguettaya, Nepal et al.2010) (Menascé and Dubey, Menascé and Dubey2007) (Bergmann, Richter, Schmitt, Stahl, and Vollrath, Bergmann et al.2001) (Li, Xu, Wu, and Zhu, Li et al.2012) (Yao, Lu, Fu, and Ji, Yao et al.2010). Using utility function, S^2R filters Web services at each level so that more costly operations (e.g., reputation calculations) are applied on a reduced number of candidate services to shorten the time and space complexity of this search process. Moreover, since service selection is an on-demand process, we apply the S^2R filters on run time.

In this section, we present an example scenario to motivate the problem and associated solution. Assume a travel planning system that is based on a service-oriented architecture (Figure 1.1.). The company provides travel planning services that include hotel booking, flight reservation, and car

rental. In addition to these reservation services, the system also provides an insurance service for the entire trip or individual travel components.

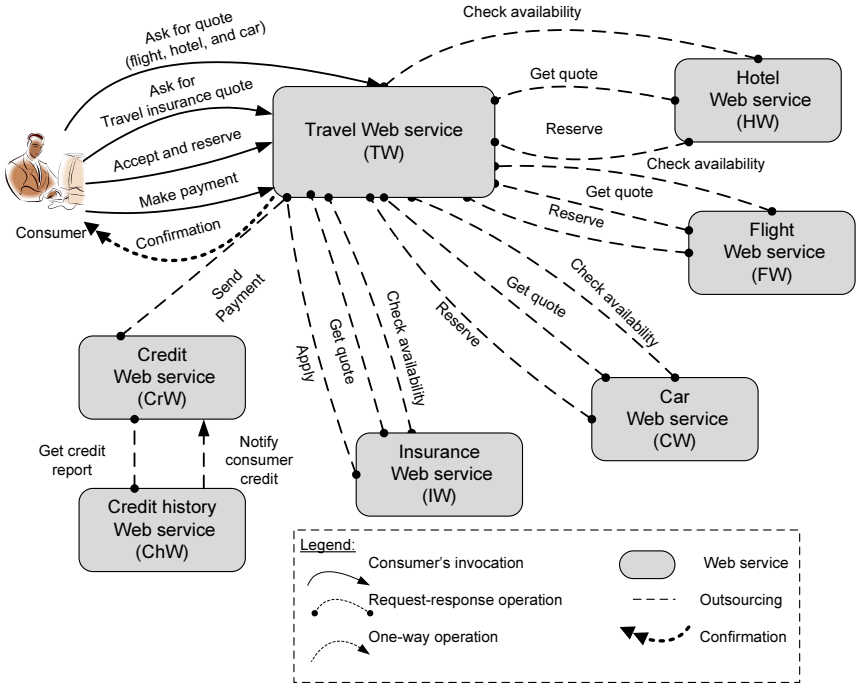


Figure 4.2: Example Scenario: A Travel Reservation System

The main Web service for the system is called Travel Web service (TW) with major operations: `Check_Availability`, `Get_Quote`, `Reserve`, `Apply`, and `Send_Payment`. TW does not implement all these functionalities by itself, rather it outsources some of the functionality to other component Web services. In Figure 4.2. we can see that component Web services (outsourced Web services) include: Hotel Web service (HW), Flight Web service (FW), Car Web service (CW), Insurance Web service (IW), and Credit Web service (CrW). The consumer invokes TW through the `Get_Quote` operation by providing the travel date, departure and arriving city information. To get the quote, TW should interact with other services (i.e., HW, FW, and CW) by checking the availability for the required dates and cities using operation `Check_Availability`. TW then

requests quotes for the available reservations through the operation `Get_Quote`. Upon receiving individual quotes from component services, TW aggregates these quotes and sends them to the consumer. At this point in the reservation process, the consumer also has the option of buying travel insurance (which TW outsources to IW). If the consumer accepts the quote, the payment process starts. TW outsources CrW to process the credit payments. CrW in turn outsources the consumer's credit check process to ChW. If the consumer's credit meets the credit score requirements, then TW makes a reservation with (HW, FW, and CW) and starts the insurance process (if consumer wills). Finally, TW notifies the consumer with the confirmation number (for flight, hotel and rental car) and sends the receipt. TW may run into some issues when it is trying to formulate this solution by outsourcing functionalities to component services. First of all, how would TW calculate the functional equivalence of two or more similar services, e.g., when TW is looking for a flight Web service, the first step is to find all the Web services that provide this functionality (i.e., resolve both syntactic and semantic equivalence). Even if TW is able to find functionally similar Web services for flight Web service, they may have different non-functional (QoS) properties (such as service A may have a response time of 3ms and service B may take 7ms to respond to user requests). Hence TW needs to differentiate among the candidate services based on the value (utility) they add to the composition. The main motivation behind S^2R is to solve the above mentioned issues while reducing the time and space complexity of this (services) search process. We believe that an efficient solution to the service selection problem is also paramount in reducing fault recovery time in SOAs, for cases where a faulty service needs to be replaced by a 'similar' one (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2011).

4.0.4 S^2R Architecture

Generally, similarity measurement consists of two components: syntactic and semantic. In syntactic similarity, we look for the similarity between data items where value of this similarity usually lies in the range [0,1]. In semantic similarity, we look for defined relationships among various terms and concepts (e.g. defined in an ontology or extracted). In S^2R, both syntactic and semantic similarity filters are applied to find a set of Web services that match users' requirements. We assume that each Web service is defined using a description language such as WSDL (Web Service Description Language) which describes the functional service properties and its interface. WSDL files are published in a service registry that allows providers to advertise general information about their Web services. This information is used by clients for discovering providers and Web services of interest. UDDI and ebXML are examples of protocols that can be used for the registration of Web services. Since UDDI is the leading specification for the development of service-based repositories or registries (Bouguettaya, Krüger, and Margaria, Bouguettaya et al.2008) we use UDDI as a registration repository, where service providers publish their WSDL files (in catalog form). UDDI is organized in form of business activity categories (including the built-in NAICS, UN/SPSC and the other user defined categories), and service providers are responsible for publishing their services in the appropriate UDDI category. Numerous Web services providing similar functionality may thus be listed under the same category in a UDDI. In S^2R, we search the UDDI and retrieve the Web services under a category and send them to the first level (Functional Context Filter (FCF)). Thus, S^2R starts by calculating the syntactic similarity for each attribute, and if a syntactic match is not found, candidate services are checked for semantic similarity. The attributes

types in FCF are: (1) syntactic attributes, which include the list of input and output parameters, the data types of the parameters, and the protocol to be used to invoke the Web service such as SOAP. (2) Semantic attributes, include the pre-conditions and effects of an operations execution.

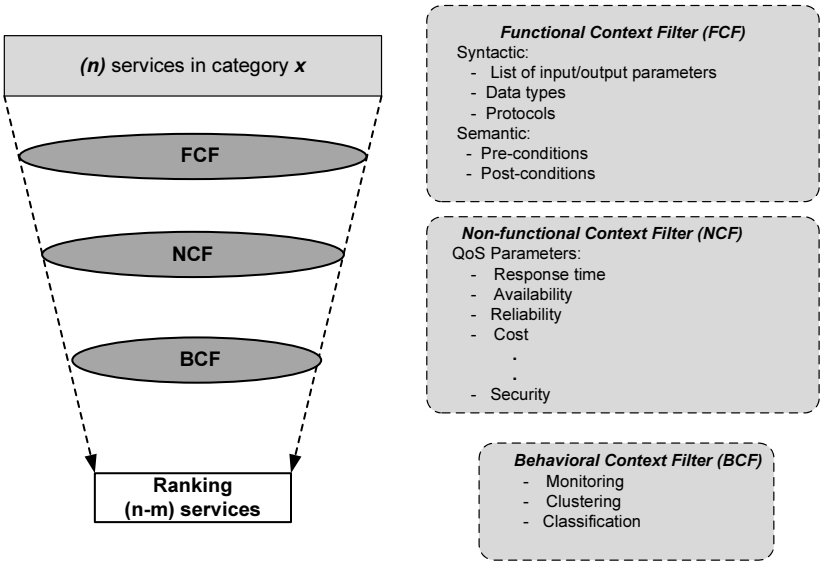


Figure 4.3: Overview of the matching levels for S^2R.

We feed the (reduced) output set from (FCF) into the second level (Non-functional Context Filter(NCF)) as in Figure 4.3. NCF is a filtering mechanism based on QoS parameters which measure the quality of a Web service. There are many parameters that can be used to measure a Web service's quality such as response time, availability, reliability, cost, security and privacy, etc. (Comuzzi and Pernici, Comuzzi and Pernici2009) (Lee, Lee2011) (Krishnamurthy and Babu, Krishnamurthy and Babu2012) (Pernici and Siadat, Pernici and Siadat2011) (Yeom, Tsai, Bai, and Lee, Yeom et al.2011). In addition, we can add any new specification to each one of these filters. After finding the providers that support the same service based on functional context (*n-m* providers), we filter them based on QoS requirements and are left with (*n-m-k* providers). In

the third level, we cluster the services based on their behavior through (Behavioral Context Filter(BCF)). Finally, we rank the candidate Web services based on their utility. In the ranking level, we rank the candidate Web services based on their utility. We divide the Web services into two sets: HighRank set and LowRank set. The HighRank set includes the Web services that have QoS values higher or equal to consumer's requested values with the constraint that the price does not exceed consumer's maximum price. However, the LowRank set includes the Web services that have QoS values lower than the requested values. In case of empty HighRank set, the first Web service in the LowRank set is considered the best candidate. Note that the first three levels (i.e., FCF, NCF and BCF) are 'context based filters'.

A context is "any information that can be used to characterize the situation of an entity. An entity is any person, place, or subject that is considered relevant to interaction between a user and an application, including the user and the application themselves" (Dey, Dey2000). Context has been used in several areas such as machine learning, computer vision, information retrieval, and decision support (Kouadri Mostéfaoui and Brézillon, Kouadri Mostéfaoui and Brézillon2006). We view context as any Web service consumer or provider-related information that enables interactions between service consumers and providers. The provider-related context contains meta-data about the provider and its service (e.g., service description, QoS, etc). Similarly, consumer-related context contains meta-data about the consumer (e.g., consumer's location, expertise level, etc). For example, a non-functional context policy may include a set of quality of service parameters (e.g., response time) associated with the service. Each context definition belongs to a certain category which can be either consumer-related or provider related. From a provider's perspective, interact-

ing with a consumer depends on the situation (i.e. current variable values) of that consumer, and vice versa. Due to space restrictions, we omit further details regarding context definition. The interested reader is referred to (Medjahed and Atif, Medjahed and Atif2007). The summary of the levels and their parameters is given in Table 4.1. In the following, we provide details for the S^2R filtering levels mentioned above.

Table 4.1: S^2R Levels

Level	Context Filter	Context Type	Parameters	Supported language	References
First	FCF	Functional Context	Syntactic and semantic	OWL-S, WSDL-S, ..	(Martin, Burstein, Mcdermott, Mcilraith, Paolucci, Sycara, Mcguinness, Sirin, and Srinivasan, Martin et al.2007), (Paolucci and Wagner, Paolucci and Wagner2006)
Second	NCF	Non-functional Context	response time, availability, reliability, cost,...	WSCL, HQML, ..	(Gu, Li, Tang, Xu, and Huang, Gu et al.2007), (Gu, Nahrstedt, Yuan, Wichadakul, and Xu, Gu et al.2001)
Third	BFC	Behavioral Context	Monitoring, clustering and classification	OWL-S	(Fogg and Eckles, Fogg and Eckles2007), (Roman and Kifer, Roman and Kifer2007), (Yahyaoui, Maamar, and Boukadi, Yahyaoui et al.2010)
Fourth	Ranking	Ranking	HighRank and LowRank	None	N/A

4.0.5 Level I: Functional Context Filter (FCF)

As mentioned earlier, the WSDL files are published in a UDDI and consist of (textual) descriptions of the Web service's operations (such as input, output, conditional output, precondition and

postcondition). While some service providers describe these functionalities in different ways (e.g. both input and precondition are described as input), so S^2R includes preconditions with inputs and postconditions with outputs.

In S^2R, we extract this and other functionalities' information from OWL-S. An OWL-S service is characterized by three types of knowledge:

1. Service profile: it describes the operation of the service. It consists of three types of information: a human readable information section which describes the service, the functions that the service provides and a list of functional attributes. For example, hotel service provides the room availability of a specific hotel, this is the human information, the functional attribute is the input, the output and any other quality of service attribute such as the response time.

2. Process-model: it describes how the service works by defining the services composition and the exact operations.

3. Service grounding: it specifies the details of how an agent can access a service (i.e., the information needed by the agent to discover the service).

If the Web service does not support OWL-S, S^2R extracts the Web service information from the UDDI using the OWL-S/UDDI mapping as shown in Figure 4.4.

In an ideal scenario we would be able to find a service that perfectly matches to user requirements. However in SOAs with numerous combination of service attributes (i.e., input, output, and operations) the chances of having such a perfect match may be slim. Thus, instead of trying to find a perfect match, we could find a Web service that fulfills the user's requirements as much as

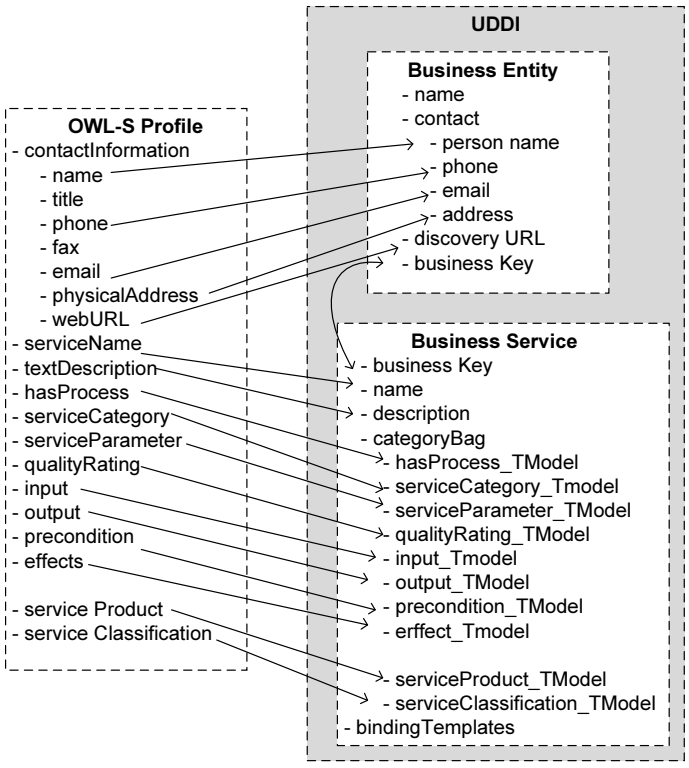


Figure 4.4: Mapping between OWL-S and UDDI constructs.

possible (i.e., Web services may provide less functionalities or may have more functionalities than requested). In S^2R, we first look for a perfectly matching service, then we increase our search to incorporate services that provide more functionalities than requested. If we cannot find any suitable candidate in the first two searches, we expand our search to include services that provide less than desired functionalities. However, in such scenarios we would need to compose multiple services to provide the requested functionality. Thus, we may have the following four scenarios (see Figure 4.5).

- Equivalent (Figure 4.5a.) Web service$_x$ and Web service$_y$ are equivalent if all operations in Web service$_x$ are exactly the same as all operations in Web service$_y$ and the number of operations

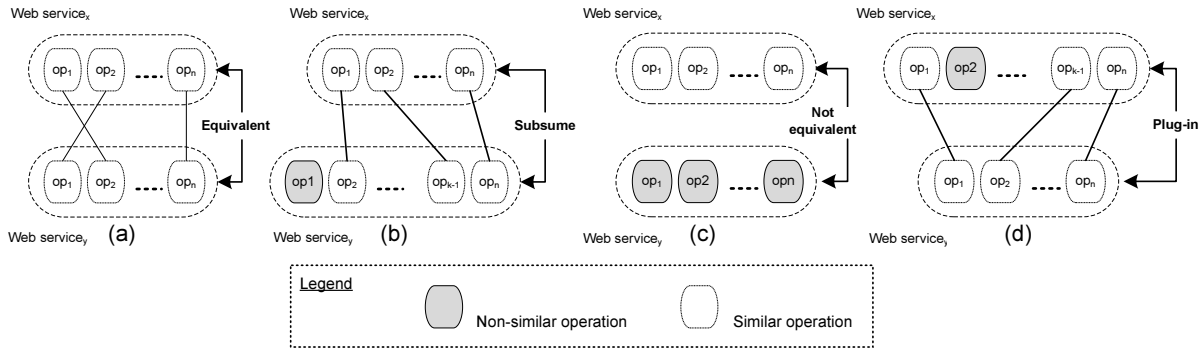


Figure 4.5: Service matching scenarios.

in Web service$_x$ is equal to the number of operations in Web service$_y$. Moreover, the inputs and outputs for each operation in Web service$_x$ are the same (names could potentially differ, e.g., cost vs price)as the inputs and outputs for each operation in Web service$_y$.

• Subsume (Figure 4.5b.) Web service$_x$ is subsumed by Web service$_y$ if all operations in Web service$_x$ are included in Web service$_y$. However, Web service$_y$ has extra inputs, outputs or operations. In this case Web service$_y$ can be counted as similar Web service to Web service$_x$ but it may request or provide extra information.

• Not-equivalent (Figure 4.5c.) Web service$_x$ and Web service$_y$ are not equivalent if all operations in Web service$_x$ do not match any operation in Web service$_y$. In this case Web service$_x$ and Web service$_y$ are totaly different.

• Plug-in (Figure 4.5d.) Web service$_y$ is plugged-in Web service$_x$ if some operations in Web service$_y$ matches some operations in Web service$_x$. In this case we need to find and compose another Web service(s) that cover the extra operations needed for Web service$_x$.

To classify any Web service under one of the matching scenarios in S^{2}R, we identify the ser-

vice operations according to the following. We consider three main types of operations: one-way, request-response, and confirmation. In one-way operations, the Web service receives a message without producing any output message (i.e., one-way communication). In request-response operation, the service receives an input message, processes it, and sends correlated output message to the sender. Confirmation operation sends an output message but does not expect to receive any more messages. `CW::Get quote::FW` is an example of a request-response operation. Its input includes departure airport, arrival airport, departure date, return date, and the number of passengers. The output message for this request-response message contains a price and room type(s). `ChW::Notify consumer credit::CrW` is a one way operation whose input contains a first name, last name, age and number of days. `TW::Confirmation` is a confirmation operation with the output of reservation details and a receipt. As we can see Request-response operations have both input and output messages. However ,One way operations only contain input messages and confirmation operations only produce output messages. Each message consists of one or more parameters called parts in a WSDL. A parameter has a name and a data type. The data type gives the range of values that maybe assigned to the parameter. The first step in finding functional equivalence among Web services is to extract this parts information from the WSDL file for the parameters and return values of operations provided by candidate Web services.

Definition 2. Two operations op_{ik} and op_{jl} match if either (1) type of message for op_{ik} = "one-way" and type of message for op_{jl} = "confirmation"; or (2) type of message for op_{ik} = "request-response" and type of message for op_{jl} ="request-response".□

Definition 3. Each Web service is accessible via operations and each operation is identified by a tuple $< Description_{ij}, Mode_{ij}, Input_{ij}, Output_{ij}, Purpose_{ij}, Category_{ij}, Quality_{ij} >$, where $Description_{ij}$ is a textual summary about the features of the operation, $Mode_{ij}$ is the type of operation (i.e., one way, response-request or confirmation), $Input_{ij}$ is the input of the operation (if it exists), $Output_{ij}$ is the output of the operation (if it exists), $Purpose_{ij}$ is the business function offered by the operation, $Category_{ij}$ describes the operation domain, $Quality_{ij}$ provides the operation's qualitative properties. □

Example: The operation `TW::Get quote::HW` in our running example. is defined by a tuple <this operation returns the price for a given date to reserve hotel, request-response, dates and number of passengers; price in dollar, bussiness.function = request for quote, hotels, Quality.price>x and Quality.security="false">.

Definition 4. $operation_{ij}$ is similar to $operation_{kl}$ or subsumed by $operation_{ij}$ if

1. $\forall x \in Input_{ij}, \exists x' \in Input_{kl} |$ x is data type compatible with x'.
2. $\forall y \in Output_{ij}, \exists y' \in Output_{kl} |$ y is data type compatible with y'.
3. ($Category_{ij}$ = $Category_{kl}$) $\vee$ ($Category_{ij} \subseteq Category_{kl}$).
4. $Mode_{ij}$ = $Mode_{kl}$.
5. ($Purpose_{ij} \equiv Purpose_{kl}$) $\vee$ ($Purpose_{ij} \subseteq Purpose_{kl}$).

6. Text matching between $Description_{ij}$ and $Description_{kl} \geq \ell | \ell$ is a pre-determined threshold.

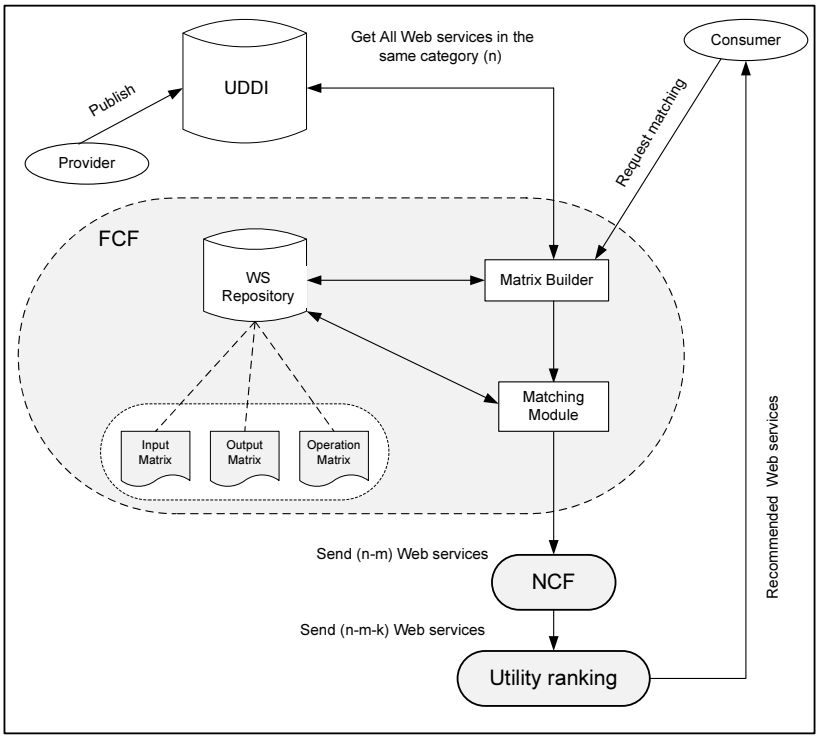


Figure 4.6: Functional Context Filter (expanded).

In FCF, we apply neighborhood calculation to find the similar Web services based on their functional properties. This filter works based on three predefined matrices: input matrix, output matrix and operation matrix. Figure 4.6 shows the steps of how FCF works. Upon arrival of a consumer request, a list of *n* services is retrieved from the UDDI under the requested category. FCF's Matrix Builder module then creates the three matrices: input, output and operation. Each matrix has $m \times n$ dimensions where *m* is the number of retrieved Web services and *n* is the number of inputs, outputs or operations for each matrix respectively. For instance, an $A_{m \times n}$ operations matrix is created as

$$A_{i,j} = \begin{cases} 1 & \text{if Web service}_i \text{ has operation}_j \text{ ;} \\ 0 & \text{otherwise.} \end{cases}$$

When the Web service$_i$ does not provide the operation$_j$, the value of $A_{i,j}$ is zero. For example, if we have non-stop operation (i.e., provide the non-stop routes) and Web service$_x$ does not provide it, we will add zero under this operation for Web service$_x$.

Table 4.2: Example of Web-operation Matrix

	Get-Quote	Get-Destination	Get-Price	Get-Time
Web service$_1$	1	1	0	1
Web service$_2$	1	1	1	0
Web service$_3$	0	1	1	0
Web service$_4$	1	1	1	1
Web service$_5$	1	1	0	0
Web service$_6$	0	1	1	1

While we are filling the matrix, the main concern is determining if the parameters of service$_x$ is the same as the parameters in the matrix. For instance, finding a flight using Web service$_x$ requires the input (airport name), but Web service$_y$ may requires the input (zip code) for the same operation. Hence it is important to find sematic similarity to address such scenarios.

S^{2}R extracts the semantic information of the candidate Web services through OWL-S. The semantics of the parameters are defined by the following attributes:

1. Consumer and provider types: the consumer and the provider should be under the same category. For example, if they provide travel services then they should be under the travel category. In case of a composite solution that has multiple categories, the *Business role* will define the category for each service.

2. Category: the category for each parameter describes the area of interest of the parameter. The category is defined by a tuple (domain, synonym, overlap). Domain gives the area of interest. For example, "travel" which takes these values from the domain in OWL-S. Synonym contains a set of alternative names for the domain name. For example, "trip" is a synonym of "travel". Overlap contains the list of categories that overlap with the current category.

3. Purpose: describes the goal of the parameter, for example, the goal of `Get-Quote` in the scenario is to return the price of the requested service.

4. Business role: the business role gives the type(category) information about a service under a certain business role. Every parameter has a well defined meaning according to the taxonomy.

5. Unit: it is the measurement unit for a parameter such as, using miles to measure the distance and dollar to measure the cost, etc.

We use Table 4.2 to illustrate matrix building. The matrix contains six Web services and four operations. The matrix dimensions are $A_{6\times4}$. The operations for Web service$_1$ are (`Get-Quote`, `Get-Destination` and `Get-Time`), the Web service$_2$ operations are (`Get-Quote`, `Get-Destination` and `Get-Price`), etc. The first step of S^{2}R is determining the inputs, outputs and operations of the Web service which based on the consumer request. If all the properties are available in the matrix then we just add the service name, and insert one under the property if the service provides it, else insert zero. However, if the property does not exist, we will edit and add the new property to the matrix. For example, one provider wants to publish Web service$_7$ which includes the

operations (`Get-Quote`, `Get-Destination`, `Get-Price` and `Get-Rate`). The first three operations exist in the matrix but the last operation is a new one. In this case we add the new property (`Get-Rate`), add one under the operations (`Get-Quote`, `Get-Destination` and `Get-Price`), so the new matrix will be as follows

Table 4.3: Example of Adding Web service to the Web-operation Matrix

	Get-Quote	Get-Destination	Get-Price	Get-Time	Get-Rate
Web service$_1$	1	1	0	1	0
Web service$_2$	1	1	1	0	0
Web service$_3$	0	1	1	0	0
Web service$_4$	1	1	1	1	0
Web service$_5$	1	1	0	0	0
Web service$_6$	0	1	1	1	0
Web service$_7$	0	1	1	1	1

FCF inserts the requirements into a vector by getting the parameters for a specific category from the service repository. It then builds a priority matrix. The priority matrix is a matrix that gives weight to each property and will move the focus towards more important operations. Based on TF-IDF (Karimzadehgan, Li, Zhang, and Mao, Karimzadehgan et al.2011), we define the priority matrix over the original matrix $A_{m\times n}$ to compute the weight of each item as:

$$w_{i,j} = \frac{A_{i,j} \times |Ws_i|}{OpM} * log\frac{A_{i,j}}{|Op_j|} \tag{4.1}$$

where $A_{i,j}$ is one if the operation *j* exists in Web service *i*, otherwise $A_{i,j}$ is zero, $|Op_j|$ is the number of times that Op_j has been used by all Web services, OpM is the number of operations in the matrix and $|Ws_i|$ is the number of operations for Web service *i*. The result after applying Equation 1. to our example matrix (in Table 4.3) will be the priority matrix in Table 4.4. The

operations that are provided most often by the Web services in the same category will have the highest weights and the operations that are provided less will have lower weights.

Table 4.4: Priority Matrix

	Get-Quote	Get-Destination	Get-Price	Get-Time	Get-Rate
Web service$_1$	0.0635	0.1111	0	0.0635	0
Web service$_2$	0.0635	0.1111	0.0794	0	0
Web service$_3$	0	0.1667	0.1190	0	0
Web service$_4$	0.0476	0.0833	0.0595	0.0476	0
Web service$_5$	0.0953	0.1667	0	0	0
Web service$_6$	0	0.1111	0.0794	0.0635	0
Web service$_7$	0	0.0833	0.0595	0.0476	0.0119

After building the priority matrix, FCF converts each row of the matrix into binary vectors, for example if the consumer request contains the operations <`Get-Quote`, `Get-Destination`, `Get-Price`, `Get-Rate`> while the available service has <`Get-Quote`, `Get-Destination`, `Get-Price`, `Get-Time`, `Get-Rate` > then the query vector of this Web service is $< 1, 1, 1, 0, 1 >$. FCF finds similar Web services based on vector similarity (Chan, Gaaloul, and Tata, Chan et al.2011) as:

$$Similarity(I, J) = |cosine\overrightarrow{V_i}, \overrightarrow{V_j}| = |\sum_{k=1}^{n}(i_k \times j_k)| \div \sqrt{\sum_{k=1}^{n} i_k^2} \times \sqrt{\sum_{k=1}^{n} j_k^2} \qquad (4.2)$$

Now, let us suppose that the consumer requests a matching for Web service that includes the operations: $<$ `Get` $-$ `Quote`, `Get` $-$ `Destination`, `Get` $-$ `Price` $>$ i.e., the vector will be $\overrightarrow{V_1} =< 1, 1, 1, 0, 0 >$ and it will compared to all vectors $\overrightarrow{V_d}$ where d $\in$ [1,m] in the matrix $A_{m\times n}$.

$$I = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}, \quad J = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \end{pmatrix}, \quad Similarity(I, J) = \begin{pmatrix} 0.6667 \\ 1.0 \\ 0.8403 \\ 0.8824 \\ 0.8403 \\ 0.6667 \\ 0.5882 \end{pmatrix} \tag{4.3}$$

If the threshold β for selecting the Web services is (0.8) then the result from FCF is the set $\{Webservice_2, Webservice_3, Webservice_4, Webservice_5\}$. In this case, we find that $Webservice_3$ and $Webservice_5$ have the same similarity value (0.8403). Notice that $Webservice_3$ does not provide the operation `Get-Quote` and $Webservice_5$ does not provide the operation `Get-Price`. In such case, we return back to the operation priority matrix which shows the priority for the operation `Get-Price` is 0.1190 and the priority for the operation `Get-Quote` is 0.0953, so we prefer $Webservice_3$ contains the higher priority operation.

4.0.6 Level II: Non-functional Context Filter (NCF)

NCF is divided into two steps: The first step checks for the service *availability*, thereby, eliminating the Web services that are unavailable. The second step checks Web service similarity based on other QoS parameters (e.g., response time, throughput, reliability, etc). We use the ping utility for the former, which has been used for Web service performance measurements (Guoping, Hui-

juan, and Zhibin, Guoping et al.2009). After determining the set of Web services that respond to the ping inquires, we start the second step where we use Context Policy Assistance (CPA) to test the similarity between the QoS parameters that are required by the consumer and the QoS parameters offered by the available Web services.

```
Context Rule  NF-cost
    Context Property  Cost
    Instance  cost_s, cost_d
    Type  NFP
    Action  matchproperity($cost_s$, $cost_d$)
      { If cost_d<= cost_s Then return true else return false }
```

Figure 4.7: Rule example.

CPA is created by the service provider and should be attached to the service. It facilitates interaction between providers and the service registry to store the context policies. For further details, the interested reader is referred to (Medjahed and Atif, Medjahed and Atif2007). A service provider may create the context specification using a context specification language such as WS-Policy. WS-Policy provides a general model and syntax to describe and communicate the policies of Web services (Erradi, Maheshwari, and Tosic, Erradi et al.2007). Each policy contains a set of rules that define the QoS requirements/capabilities of the Web services. A sample rule is shown in Figure 4.7. where a context rule is identified by a name to specify the property. In this rule we need two instances: the consumer's cost and the provider's cost. The type of this policy is NFP (non-functional policy) and it compares the cost between the two parties to determine if they are compatible. We use these policies to determine if two Web services are similar based on the QoS parameters they both share (Alrifai, Skoutas, and Risse, Alrifai et al.2010).

Definition 5. QoS vector description: it is an extendable vector used to define QoS parameters

of the provider and the consumer. It is expressed as: QoS = $< QoS_1, QoS_2, ..., QoS_n >$, n $\in$ R, where QoS_n indicates nth QoS attributes and QoS$_i$ where i $\in$ [1,n] is equal to {Availability, Cost, Response time, Error rate, Throughput, Reliability, Reputation, and Security.}.□

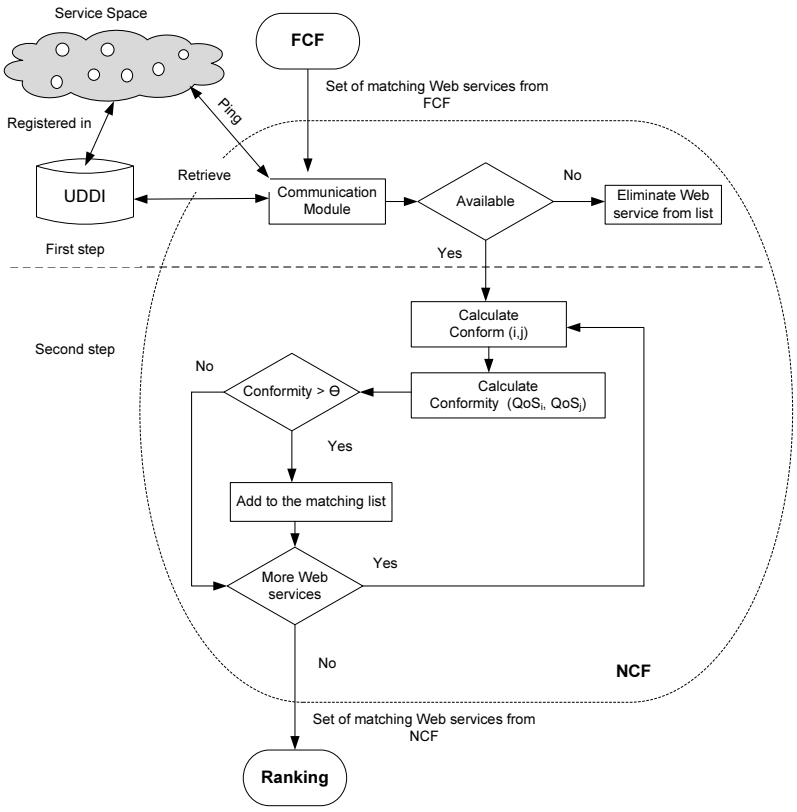


Figure 4.8: Non-functional Filter (expanded).

QoS parameter matching is done as:

1. Convert the parameters into QoS vector descriptions. Then, we have one vector for the consumer request:

   $< ConQoS_1, ConQoS_2, ..., ConQoS_n >$ and multiple vectors for the provider offerings:

   $< Pro_1QoS_1, Pro_1QoS_2, ..., Pro_1QoS_n >$ ... $< Pro_kQoS_1, Prov_kQoS_2, ..., Prov_kQoS_n >$.

Here we assume that the providers provide trusted information for the QoS values (Sherchan, Nepal, Hunklinger, and Bouguettaya, Sherchan et al.2010). There are three different cases in converting the QoS parameters: the consumer vector is greater than the provider vector, the consumer vector is less than the provider vector, or the consumer vector is equal to the provider vector. In the first case we add zeros at the end of the consumer vector and in the second case we add zeros at the end of provider vector.

2. Create a new vector called conform with length equal to the max length of consumer and provider vectors. For each element in the vector use the polices to compare the conditions, and if the condition is met then add one to the conform vector else put zero. At the end of this step we will have the vector $< conform_1, conform_2, ..., conform_n >$.

3. Calculate the conformity degree between the services for the consumer QoS_i and the provider QoS_j as:

$$Conformity(QoS_i, QoS_j) = \sum_{q=1}^{z} Weight_q * conform_q \tag{4.4}$$

where i,j are Web services, z is the maximum length of parameters, i.e., $z = max(|QoS_i|, |QoS_j|)$, and $Weight_q$ is the weight assigned to each QoS parameter.

Consumers may have different expectations about the conformity degree of their services. For this purpose, they provide a conformity threshold θ $(0 < \theta \leq 1)$. In NFC (Figure 4.8), we find all Web services j where the conformity degree (QoS_i, QoS_j) is greater than θ, which are then passed to the ranking level. The conformity threshold is given by the consumer as a part of his profile, while the QoS weight is created automatically by the system based on the level of the consumer's

expertise. In S^2R, we defined three types of consumers: expert, regular and normal consumers. The expert consumers are knowledgable about meaning of all QoS parameters and they may assign the desired value for the QoS parameters for specific services. The regular consumers have some knowledge about the QoS parameters, and they may assign values to some QoS parameters and leave the other parameters without weights. In this case, the system predefined weights are used for unassigned parameters. The normal consumers do not have any knowledge about the QoS parameters that the system assigns weights for all parameters.

The main benefit of categorizing the consumers into these types is to let the expert consumer participate in making a decision by providing weights for each QoS parameter. However, the system will provide all the QoS attribute weights for other categories of the consumers. In essence, consumer categories are determined based on the assigned values for the current request.

Example: Suppose that Web service HW is one of the candidate Web services as a result of FCF. Let it have the following QoS vector: $<$ price = \$50; response-time= 60 sec; error-rate = 0.01; security="false"$>$. On the other hand, the consumer (Web service$_x$) QoS vector is as following: $<$ price $\leq$ \$70; response-time $<$ 90 sec; error-rate $<$ 0.05; reliability= 0.80; security="true"$>$. The first step is building the conformity vector: $< 1, 1, 1, 0, 0 >$. The first three values of the vector are equal to one because the conditions are met between Web service$_x$ and Web service$_y$. However, since the values of reliability and security do not match, 0's are appended. The conformity degree based on individual QoS parameter weights is then assessed. Assume that the consumer provides

the weights as $< 0.3, 0.2, 0.4, 0.1, 0.3 >$, then:

$$Conformity(QoS_x, QoS_y) = \sum_{q=1}^{5} < 0.3, 0.4, 0.1, 0.1, 0.1 > * < 1, 1, 1, 0, 0 >= 0.8 \qquad (4.5)$$

4.0.7 Level III: Behavioral Context Filter (FCF)

A behavior observation is an observation of a Web service quality for one interaction. This observation is computed based on the distance between the score of a Web service and the average of quality attribute values of Web services belonging to the same domain[1].

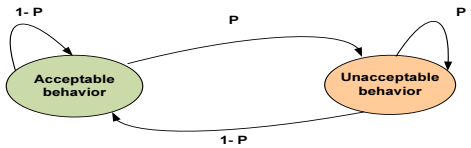


Figure 4.9: Behavioral patterns using HMM.

For classification purposes, we cluster the Web service into two states: acceptable (Acc) and unacceptable (Ucc) behavior (Figure 4.9). The degree of acceptance is based on the conformity of QoS between the service provider and service consumer. The *experiences* are evaluated as a ratio of the number of times in the acceptable state, divided by the total number of times the service was invoked. Each time the composition orchestrator invokes a service, it records the state of that service (acceptable or unacceptable) along with the time of invocation. Let the vector V = the service behavior profile, then to asses the probability that $Service_i$ will be in the acceptable state in the next time instance:

[1] A domain gathers services having the same functionality

$$P(Acc|V) = P(Acc|Ucc) + P(Acc|Acc) \tag{4.6}$$

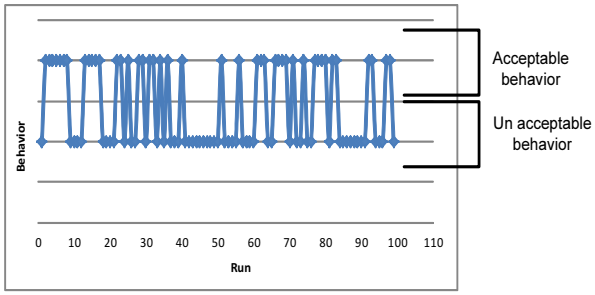


Figure 4.10: Web service behavioral pattern and clusters.

Hidden Markov models have successfully been applied to various pattern recognition problems (Salfner, Schieschke, and Malek, Salfner et al.2006). We also use an HMM to predict services' behaviors. We use the Forward algorithm to train the model and calculate the sequence likelihood. The sequence likelihood is the probability that model χ can generate observation O. Assume that we train two sequences: acceptable sequence χ and unacceptable sequence χ'. Now, we need to determine whether a given observation is acceptable or non-acceptable. The new observation is defined using the vector O = [o_0, o_1, ... o_h]. This is done using Bayes decision theory to classify the observations (see Figure 4.10).

The *Forward algorithm* is based on a forward variable $\alpha_t(i)$ denoting the probability of subsequence [o_0, o_1, ... o_t] and the fact that the stochastic process is in state i at time t:

$$\alpha_t(i) = Pr(o_0 o_1 .. o_t, P_t = p_i|\chi) \tag{4.7}$$

$\alpha_t(i)$ can be computed by the following recursive computation scheme:

$$\alpha_0(i) = \chi_i \; g_i(o_0) \tag{4.8}$$

Each time the process enters a state, one observation is generated according to the probability ($g_i(o_z)$), where $\alpha_t(i)$ is the probability of the entire sequence and the fact that the stochastic process is in state i at the end of the sequence, sequence likelihood Pr($O|\chi$) can be computed as:

$$Pr(O|\chi) = \sum_{i=1}^{2} \alpha_t(i) \tag{4.9}$$

After behavior classification, we rank the Web services based on the utility in L4.

4.0.8 Level IV: Web service Ranking

In this level, S^{2}R ranks the Web services based on the range compatibility of the QoS parameters. We use weighted sum filter function after converting the QoS parameters into a range vector in the format of the component vector description.

Definition 6. The component vector description is expressed as:

$< (QoS1, QoS1_{min}, QoS1_{max}), (QoS2, QoS2_{min}, QoS2_{max}), ..., (QoSn, QoSn_{min}, QoSn_{max}) >$, n $\in$ R, where QoSi$_{\in[1,n]}$ is the best QoS value for parameter i, $QoSi_{min}$ is the minimum acceptable value for parameter i, and $QoSi_{max}$ is the maximum acceptable value for parameter i, then $QoSi_{min} \leq QoSi \leq QoSi_{max}$.$\square$

Each vector is accompanied by a decision model, i.e. ranges of all the QoS parameters as well

as their respective priorities also known as the weights. The ranking will be based on the matching degree (i.e., how near the QoS parameter that is provided by the provider is to the QoS parameters required by the consumer). Note that we can use the provider QoS values in one of two ways:(i) QoS values as advertised by the service provider, or (ii) QoS values obtained using behavior monitoring through the community (i.e., provider reputation). The best ranked Web service will be the Web service that is closest to the 'best' value and the worst ranked will be the Web service that is the farthest from the 'best' value. However, if a Web service provides a larger value than the best value but has a lower cost associated to it, then the system will give this Web service a higher rank.

Definition 7. $\forall$ Web services $WS_i \in \omega$, where ω is the set of all Web services which are similar (functional and non-functional) to the requested service:

$$WebServiceRankSet = \begin{cases} HighRank & \text{if } WS_i \geq \mu \wedge cost \leq \wp; \\ LowRank & \text{if } WS_i < \mu. \end{cases}$$

where μ is the best value of QoS_j that is provided by the consumer from the vector $< (QoS_j, QoS_{jmin}, QoS_{jmax}) >$, and $\wp$ is the acceptable cost by the consumer. $\square$

We assume that all the participating Web services are able to articulate their objectives and prioritize them (Ackoff, Ackoff1978). The articulation and prioritization of objective values is well accepted in multi-attribute situations and operations research (Chandra, Ellis, and Vahdat, Chandra et al.2000) (Faratin, Sierra, and Jennings, Faratin et al.2002) (Resinas, Fernandez, and Corchuelo, Resinas et al.2012). The consumer determines/assigns a priority for each QoS parameter (e.g., the price of the service is more important than its execution time). In our method, we covert these

priorities into a weighted vector to compare the consumer requirements with the provider offer. All the Web services conform to some constraints in the solution. For instance, any QoS vector cannot have a negative value (as shown by Equation 4.10), and the QoS values lie between the maximum and minimum allowable values set by the consumer Web service (as shown by Equation 4.11).

$$X_j \geq 0 \quad and \quad Y_{ij} \geq 0 \tag{4.10}$$

$$X_{j(min)} \leq X_j \leq X_{j(max)} \quad and \quad Y_{ij(min)} \leq Y_{ij} \leq Y_{ij(max)} \tag{4.11}$$

The utility function is a multi-step calculation that evaluates the degree of matching between the Web services. A weighted sum approach is used to combine these multiple QoS parameters. We use a distance function to measure the difference among the proposed solutions of both the consumer and provider Web services. Thus, lower utility values are desired as they translate to lesser mismatch among the services. Similarly, lower values translate to higher ranks for the solutions among the solution space. The utility value of a match is calculated as follows

$$\Delta_{ij} = \frac{|X_j - Y_{ij}|}{X_j} \tag{4.12}$$

$$r_j = \sum_{j=0}^{n} (WX_j * \Delta_{ij} + WY_{ij} * \Delta_{ij}) \tag{4.13}$$

$$R_s = min \sum_{j=0}^{G} (r_j) \tag{4.14}$$

S^2R technique succeed in finding the best performance similar Web services in the Cloud. Since user requirements for cloud services vary, service providers have to ensure that they can be flexible in their service delivery while keeping the users isolated from the underlying infrastructure. On the other hand, the service consumers require faster response time which may be achieved by distributing requests to multiple Clouds in various locations at the same time. This creates the need for finding similar services or applications on different clouds. There are many challenges involved in finding such services such as the functional and non-functional attributes.

Cloud computing could also be refereed to as service-oriented computing (SOC) paradigm. Web services are self-described, self-contained and platform-independent computational elements that can be published, discovered, and composed using standard protocols, to build applications across various platforms and organizations in a dynamic manner. With the increasing agreement on the functional aspects of Web services, such as using WSDL (Booth and Liu, Booth and Liu2006) for service description, SOAP (SOAP, SOAP2007) for communication and WS-BPEL (WSBPEL, WSBPEL2005) for composing Web services etc., the research interest is shifting towards the non-functional aspects of Web services (Papazoglou, Pohl, Parkin, and Metzger, Papazoglou et al.2010) (Buyya, Yeo, Venugopal, Broberg, and Brandic, Buyya et al.2009).

We present an example scenario to motivate the problem and associated solution. Figure 4.11. shows a typical service auction scenario in a cloud computing environment. In the example shown, broker conduct an auction to match bids of multiple resources advertised by different providers.

These providers advertise their resources with the required price. The consumers then submit their bids to show the degree of interest in the advertised resources. The bids from the consumers are queued in the database by *Job schedule services* which will help in calculating the winning bid. On the other hand, the resources are indexed and stored in the database by *Catalogue services*. After that, the *Trading broker service* coordinates the matching of resources and bids, and trading between auction participants. At the end of the auction the broker decides the winners and sends the reservation requests to the *Reservation service*. Then the reservation service informs the resource providers and consumers about the final result (i.e., who won the bid). The payment processing takes place through the *Accounting service*.

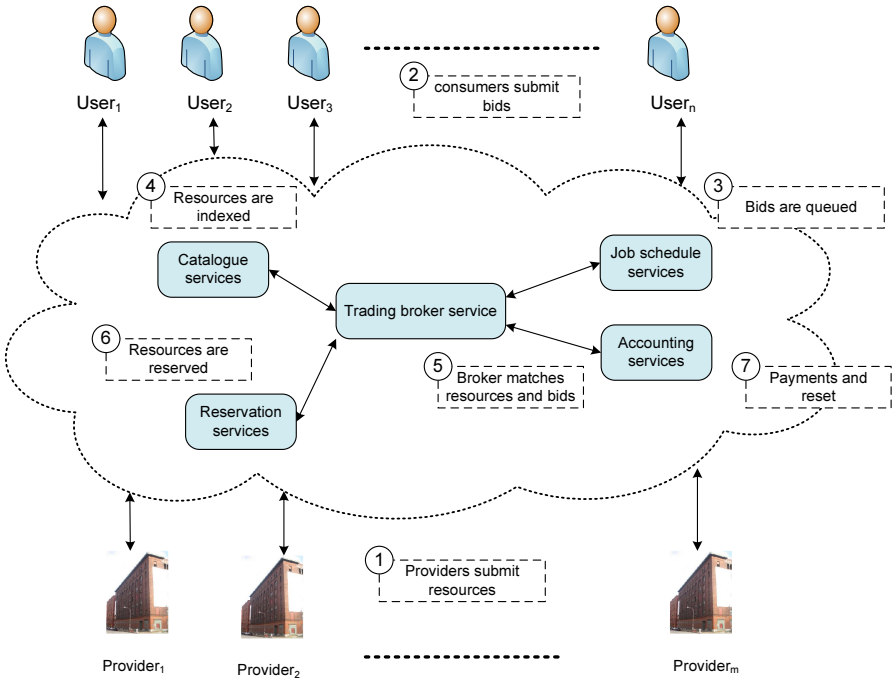


Figure 4.11: A trading scenario in the cloud

In Figure 4.11, we focus on five main services: Job schedule services, Catalogue services, Trading broker service, Reservation service and Accounting service. Job schedule service is the service

that stores the bids coming from the consumers and sorts them by date, time and price. Catalogue service stores the resources advertised by the providers and indexes them to reduce the complexity for the bid matching process. Trading broker service is the service that manages the auction and matches the bids with the required price. Reservation service is the service that reserve a specific resource to the winner of the auction and informs results to the participants of the auction process. Finally, accounting service is the service that is responsible for processing payments: check if the consumer's credit meets the credit score requirements, make a payment and send the results to the user. The above scenario has many challenges. First and the foremost would be calculating the functional equivalence for the two or more similar services, e.g., when looking for an accounting service, the first step is to find all the Web services that provide this functionality (i.e., resolve both syntactic and semantic equivalence). Even if it is able to find functionally similar Web services for accounting service, they may have different non-functional (QoS) properties (such as service A may have a response time of 3ms and service B may take 7ms to respond to user requests). Hence we needs to differentiate among the candidate services based on the value (utility) they add to the composition. The main motivation behind our technique is to solve the above mentioned issues while reducing the time and space complexity of this (services) search process. We believe that an efficient solution to the service selection problem is also paramount in reducing fault recovery time in SOAs, for cases where a faulty service needs to be replaced by a 'similar' one (Alhosban, Hashmi, Malik, and Medjahed, Alhosban et al.2011).

CHAPTER 5

FAULT MANAGEMENT PROPAGATION

In this chapter, we focus on faults that propagate from participants to composite services. We refer to these faults as bottom-up. Some of the examples of bottom-up-faults may include, a hardware failure occurring in the participant service deeming it unreachable by the composite system, a scheduled maintenance downtime for the participant's provider, change in participants interface (e.g. changing an optional parameter to mandatory in WSDL specification). Composite services therefore need to detect and handle any faults as soon as possible, to avoid any run-time failures or service outages in overall system.

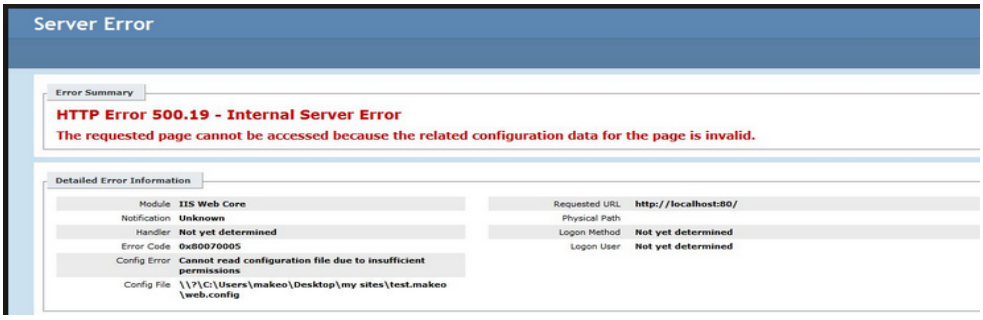


Figure 5.1: HTTP message example.

Web services generally use HTTP as the underlying message transport (example of such messages in Figure 5.1). Hence, they are either guaranteed message delivery or notified if a message was not delivered (e.g., because of a server unavailability). This adds another layer of information to the system and hence could differentiate among a lost message vs. service outages. However, this also means that composite services become aware of a fault only at the time they interact with their participants and *not* at the time that fault occurred. Hence, the current state of a participant

service is a function of last interaction that occurred among the participant and composite service, which may or may not represent the current state of the service. This may decrease the availability of composite services. Besides, users' requests are pending as long as the composite service did not recover from the fault (e.g., by replacing the faulty participant with an equivalent one). This calls for a framework in which composite services are able to detect and handle bottom-up faults as soon as those faults occur in their participants and once detected should recover gracefully from these.

In essence, the recovery mechanism should allow the composite system to continue providing its core services to the customers (e.g., through service change monitoring). Service changes may, for instance, originate from the introduction of a new functionality, the modification of existing functionality to improve performance, or the inclusion of new regulatory constraints that require service behavior to be altered. Such changes should not be disruptive, i.e., requiring radical modifications in the very fabric of services, or the way that business is conducted (Andrikopoulos, Benbernou, and Papazoglou, Andrikopoulos et al.2008). Since routine change increases the propensity for error, one needs to know why a change was made, what are its implications, and whether the change is complete. In a Web services environment, changes only affect the Web service provider's system. Typically Web service consumers do not immediately perceive the upgraded process, particularly the detailed changes of Web services. Hence, Web service based applications may fail on the Web service client side due to changes carried out during the provider service upgrade. In order to manage changes as a whole, the Web service consumers have to be taken into consideration as well, otherwise changes that are introduced at the service producer side can create severe

disruption.

In addition, to control service evolution, a designer must know why a change was made, what its implications are, and whether the change is consistent. Eliminating spurious results and inconsistencies that occur because of uncontrolled changes is necessary for services to evolve gracefully, ensure stability, and handle variability in their behavior. We can classify the nature of service changes depending on their causal effects: *Shallow changes*. These are small-scale incremental changes localized to a service or restricted to the services clients. *Deep changes*. These are large-scale transformational changes cascading beyond a services clients, possibly to entire value chains (end-to-end service networks). While both shallow and deep changes need an appropriate versioning strategy, deep changes further introduce several intricacies of their own and require the assistance of a change-oriented service life cycle to allow services to react appropriately to changes as they occur (Andrikopoulos, Benbernou, and Papazoglou, Andrikopoulos et al.2008).

In this chapter, we introduce a framework for managing and recovering from bottom-up faults in composite services. The proposed framework (extending (Medjahed and Malik, Medjahed and Malik2011)) uses *soft-state* signaling to propagate faults from participants to composite services and uses a semantic rule based recovery mechanism to employ the traditional system recovery strategies. Soft state denotes a type of protocols where state (e.g., whether a server is alive) is constantly refreshed by periodic messages; state which is not refreshed in time expires. This is in contrast to hard-state where installed state remains installed unless explicitly removed by the receipt of a state-teardown message. Advantages of the soft-state approach include implicit error recovery and easier fault management resulting in high availability (Alhosban, Hashmi, Malik,

and Medjahed, Alhosban et al.2011). Soft state was introduced in the late 1980s and has been widely used in various Internet protocols (e.g., RSVP). However, to the best of our knowledge, this work is the first to use soft-state for fault management in composite services. We then employ a semantic rule based approach to apply multiple recovery strategies to ensure the stability of the system. We use ECA based rules that allow system designers the flexibility and consistency to device multiple recovery strategies. The strategies could be devised at both the system composition time or during the execution of the system during the optimization/enhancement phase. The major contributions of the work can be summarized as follows: First, we introduce a bottom-up fault model for composite services. The model includes a taxonomy of bottom-up faults, a definition of state for composite services, and a peer-to-peer topology for state propagation. Second, we propose a soft-state based framework for bottom-up fault management in composite services. The framework includes: protocols for fault detection (push and pull), propagation (pure soft-state and soft-state with explicit removal), and reaction (policy-based). Finally, we conduct a comprehensive set of experiments to assess the performance and applicability of the proposed framework.

5.1 Motivation

To better motivate the need for bottom-up fault management in composite services, consider the scenario depicted in Figure 5.2. The scenario shows two composite services; Travel Composition, and City Tour Composition (CS_1 and CS_2 respectively) that outsource some of their functionality to other participants. For instance, at the reception of the travel planning request (step 1), CS_1 invokes WS_1 to book the flight (step 2). Then, it invokes WS_2 to book the hotel (step 3). Finally,

it invokes WS_3 to book a car from the airport to the hotel (step 4), and returns the result to the user (step 5). Similarly, CS_2 receives a city tour request from the user (step a). It first invokes WS_3 to book a car for site-seeing purposes, etc (step b). Then, it invokes WS_4 to book a movie ticket (step c), and WS_5 to book a museum ticket (step d). Finally, it returns the result to the user (step e). Note that WS_3 is being shared by two different compositions.

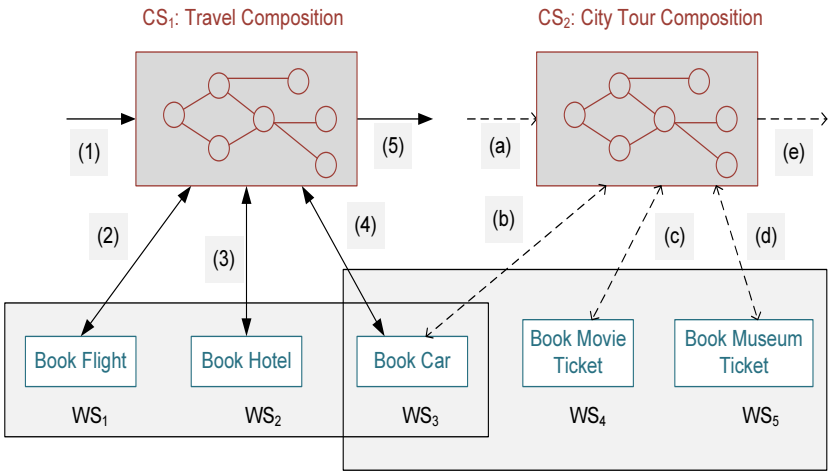


Figure 5.2: Reference Scenario.

Let us consider the case of a server failure in WS_1. When CS_1 receives a user request, it invokes WS_1 (step 2), but since WS_1 is no longer able to process incoming requests, CS_1 is notified by the underlying HTTP server (if HTTP is used as a transport protocol) that the message was not delivered to WS_1. CS_1 handles this situation (e.g., as part of its exception handling code) by returning a run-time error message to the user (e.g., "your request cannot be executed at this time"). Note that both CS_1 and WS_1 are autonomous and offered by different providers. Hence the CS_1 provider is not necessarily informed by the WS_1 provider about the unavailability of WS_1; WS_1 provider may even not know that CS_1 is planning to use WS_1 as a participant. Let us now

consider the case where the WS_3 provider changes the WSDL specification of WS_3 by adding a new parameter to its input message. Let us assume that CS_1 and CS_2 receive user requests. Since CS_1 and CS_2 are not aware of WS_3's update, they will forward users' requests to WS_3 using an obsolete message signature. WS_3 will hence send back a run-time error message to CS_1 and CS_2. As in the previous case, both composite services will in turn send error messages to their users.

The above mentioned cases illustrate the following:

- The composite services returned run-time errors to users because of the inability of those services to promptly detect faults in their participants. Such situations may be unacceptable in applications such as disaster management (e.g., unavailability of an emergency service), supply chain (e.g., a supplier running out of stock for a given product because of a strike), and real-time systems (e.g., failure of a computing resource such as processor in a real-time application).

- The faults that composite services need to deal with are physical, such as a server failure in the first case, as well as logical, such as an update of a service policy (i.e., WSDL specification) in the second case.

In the rest of this chapter, we present a framework that enables composite service to automatically manage faults in their participants as soon as those faults occur, as opposed to the participant invocation time.

5.2 Fault Model

In this section, we describe our model for bottom-up fault management. We first provide a categorization of bottom-up faults. Then, we define the notion of a participant's state. Finally, we introduce a peer-to-peer topology for bottom-up fault management.

5.2.1 Bottom-up Fault Taxonomy

A fault management approach must refer to a taxonomy that describes the different types of faults that composite services are expected to be able to manage. We identify two types of bottom-up faults: physical and logical (Figure 5.3). *Physical faults* are related to the infrastructure that supports Web service. A node fault occurs if the servers (e.g., application server, Web server) hosting a participant are out of action. *Logical faults* are initiated by service providers; this is in contrast to physical faults which are out of service providers' control. We categorize logical faults as *status change, policy change,* and *participation refusal*. *Status change* occurs if the service provider explicitly modifies the availability status of its service. The status may be changed through freeze or stop. In the *freeze* fault, providers shut down their services for limited time periods (e.g., for maintenance, unavailability of a product in a supply chain's provider). In the *stop* fault, providers make their services permanently unavailable (e.g., a company going out-of-business). *Participation refusal* occurs if a service is not willing to participate in a given composition. *Policy change* occurs if the provider updates one of its service policies. We adopt a broad definition of policy, encompassing all requirements under which a service may be consumed. Policies are specified in XML-based Web service languages/standards (e.g., WSDL, WS-Security) (Alonso,

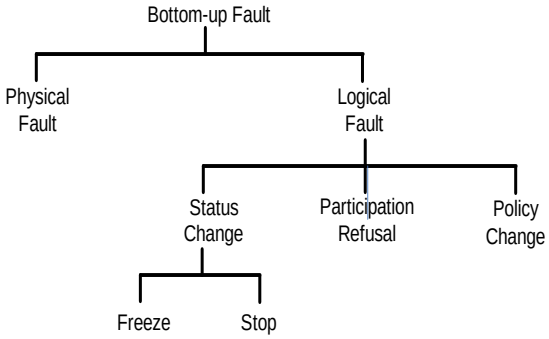


Figure 5.3: Bottom-up Fault Taxonomy.

Casati, Kuno, and Machiraju, Alonso et al.2004). Changes in the policies of a participant WS_i may impact the way a composite service CS_j interacts with WS_i. Hence, they should be considered as logical faults (e.g., as shown in the running example).

5.2.2 State of a participant service

Soft-state signaling enables the propagation of bottom-up faults from participants to composite services. The main idea of this class of signaling is that the *state* of each participant is periodically sent to the composite service. The composite service will then use the received state to determine whether there was any physical or logical fault in the participant. Several questions need to be tackled when designing a soft-state protocol: what is the definition of a state? And how is the state computed? We will give answers to these questions in the rest of this chapter.

The proposed framework must deal with all types of faults depicted in Figure 5.3. Physical faults are detected by composite services in an *implicit* manner; if a node fault occurs at a participant, then the composite service will not receive a state from that participant. The participation

refusal fault is *explicitly* communicated by participants if they are not willing to be part of a composition. Status change faults are detected either *implicitly* (pure soft-state protocol) or *explicitly* (soft-state with explicit removal protocol). Finally, policy change faults are transmitted as part of the *participant's state* and hence, they are detected in an *explicit* manner. We will give below a definition of *state* and the way it is computed.

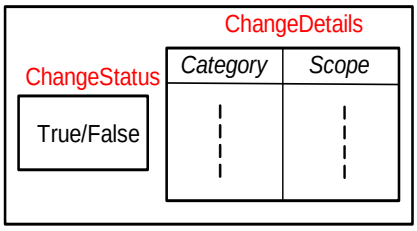


Figure 5.4: State of a Participant Service.

To keep track of policy changes, each participant WS_i maintains a data structure called $State_i$ (Figure 5.4). $State_i$ is defined by two attributes: *ChangeStatus* and *ChangeDetails*. *ChangeStatus* is equal to True if policy changes have been made to WS_i. Several changes may occur in WS_i during a time period; details about these changes are stored in the *ChangeDetails* set. Each element of this set represents a policy change; it is defined by a couple (C,S) where C is the *category* of the policy and S is the *scope* of the change. The initial values of *ChangeStatus* and *ChangeDetails* are False and $\emptyset$, respectively. The following definition summarizes the properties of $State_i$ maintained by a participant WS_i.

Definition

The state, denoted $State_i$, of a participant WS_i is defined by (ChangeStatus,ChangeDetails) where:

- ChangeStatus = True $\Longleftrightarrow$ changes have been made to WS_i.

- ChangeDetails = {(C,S) / C and S are the category and scope of a change in WS_i}.
- Initially do: $State_i$.ChangeStatus = False and $State_i$.ChangeDetails = $\emptyset$.
- At the occurrence of a change (C,S) in WS_i do: $State_i$.ChangeStatus = True, $State_i$.ChangeDetails = $State_i$.ChangeDetails $\cup$ {(C,S)}.□

A policy *category* refers to the type of requirements specified by a policy. We adopt the policy categorization defined in (Garlan and Schmerl, Garlan and Schmerl2002) which classifies policies as functional, non-functional, value-added, and specialized. *Functional* policies describe the operational features of a Web service (e.g., in WSDL). *Non-functional* policies include parameters that measure the quality of the service (e.g., response time). *Value-added* policies provide "better" environments for Web service interactions. They refer to a set of specifications for supporting optional (but important) requirements for the service (e.g., security, privacy, conversation). A *specialized* policy defines requirements that are specific to an application domain. Shipping and billing are examples of specialized policies in business-to-business e-commerce. The *scope* of a change defines the subject to which that change was applied. It includes details about (i) the location of the modified policy specification and (ii) the element that has been updated within that specification. The specification location is given by the URI of the XML file that stores the specification. The updated element is identified by the XPath query of that element within the specification. For instance, let us consider the following WSDL file (Figure 5.5) located at "http://www.ws.com/stockquote.wsdl".

Let us assume that the name of the operation "getQuote" has been modified in the WSDL document. The category and scope of the change can then be defined as:

- Category = (Functional,WSDL).

```
<definitions>
  <types>  <!-- XML Schema --> </types>
  <message name="getQuote_In"> ….
  <message name="getQuote_Out"> …
  <portType name="StockQuoteServiceInterface">
    <operation name="getQuote">
      <input message="getQuote_In" />
      <output message="getQuote_Out" />
    </operation>
  </portType>
```

Figure 5.5: WSDL file example.

- Scope = (URL,Q) where:
  - URL = "http://www.ws.com/stockquote.wsdl"
  - Q = "definitions/portType/operation/@name"

5.2.3 Fault coordinators

In the proposed framework, fault management is a collaborative process between architectural modules called *fault coordinators*. Each Web service (participant or composite) has one coordinator associated to it. This peer-to-peer topology distributes control and externalizes fault management, hence creating a clear separation between the business logic of the services and fault management tasks.

We define two types of coordinators (Figure 5.6): *soft-state senders* (SS-S) and *soft-state receivers* (SS-R). Each participant (resp., composite service) has a sender (resp., receiver) attached to it. A sender SS-S$_i$ maintains the $State_i$ data structure. To keep track of its receivers, SS-S$_i$ maintains a Receivers(SS-S$_i$) data structure. If WS_i (attached to SS-S$_i$) participates in CS_j (at-

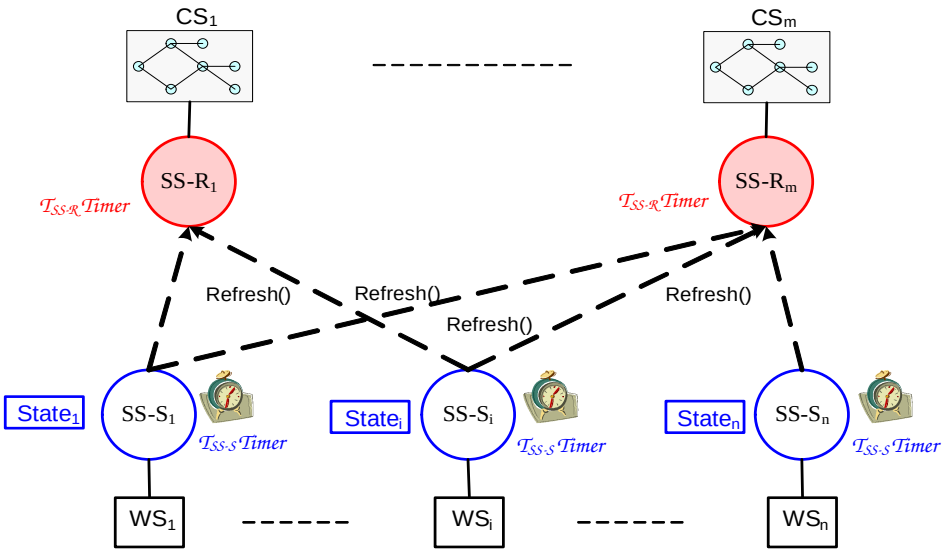


Figure 5.6: Fault Coordinators.

tached to SS-R$_j$) then SS-R$_j$ $\in$ Receivers(SS-S$_i$). SS-S$_i$ periodically sends $State_i$ to its receivers via Refresh() messages. The refresh period is determined by the t_{SSS} timer maintained by SS-S$_i$. A receiver SS-R$_j$ maintains two data structures: Senders(SS-R$_j$) and t_{SSR}. Senders(SS-R$_j$) is the set of senders from which SS-R$_j$ expects to receive Refresh(). If WS_i participates in CS_j then SS-S$_i$ $\in$ Senders(SS-R$_j$). t_{SSR} is a timer used by SS-R$_j$ to process Refresh() messages received from its senders.

Bottom-up fault management involves three major tasks: fault detection, fault propagation, and fault reaction. The sequence diagram in Figure 5.7 depicts the relationship between these tasks. First, SS-S$_i$ detects faults that occurred in the attached WS_i. This task is collaborative between WS_i and SS-S$_i$, as stated in Figure 5.6. Messages may be passed to/from WS_i during this task. Then, SS-S$_i$ propagates the fault to SS-R$_j$. As for fault detection, this task is collaborative between the sender and receiver. Finally, SS-R$_j$ and/or CS_j execute appropriate measures to react to the

fault. Details about fault detection, propagation, and reaction follow in the upcoming Sections.

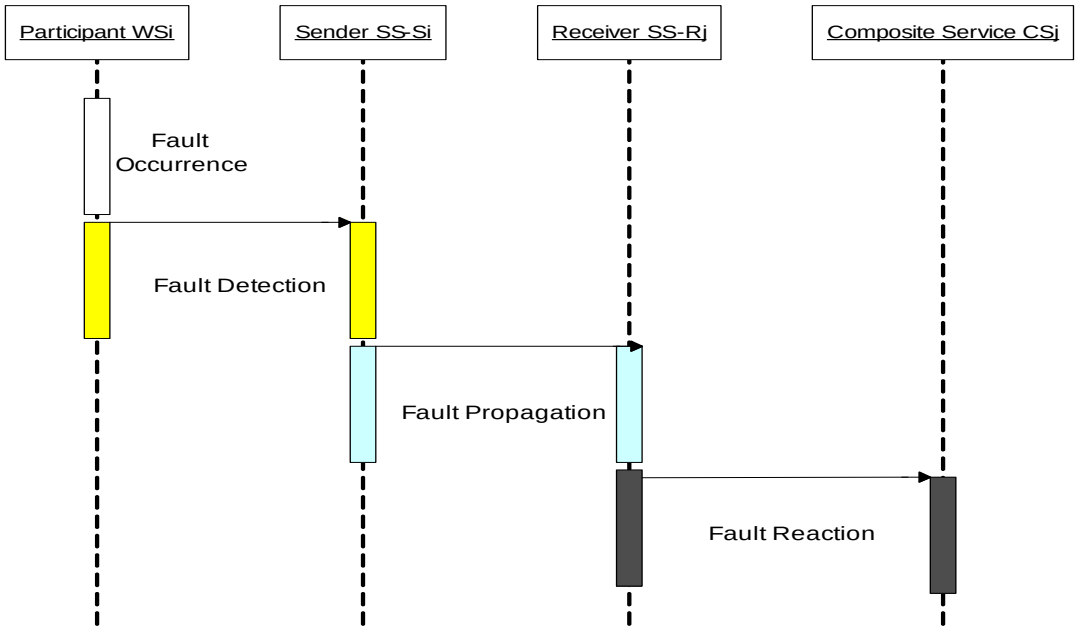


Figure 5.7: Sequence Diagram for Bottom-up Fault Management.

5.3 Fault Detection and Propagation

Each fault coordinator is itself deployed as a Web service on the same node and servers (e.g., Web server) as the attached service. Hence, identifying physical faults in a coordinator is equivalent to identifying physical faults in its attached service: if a physical fault affects a service (participant or composite), it also affects its coordinator; and vice versa.

The way participation decision is made varies from a participant to another. We give below four scenarios on how such decisions could be made:

1. A service load balancer may check that the server workload will not exceed a given threshold if the service participates in a new composition. The threshold could, for instance, be defined

to maintain a minimum quality of service.

2. A policy compatibility checker may also verify the compliance of the service policies with the composite service policies. For instance, the checker could compare privacy policies to make sure that the participant and composite service have compatible expectations and requirements about the privacy of their data.

3. A service reputation manager may verify that the reputation (however reputation is defined) of the composite service is higher than a minimum value defined by the participant's provider.

4. A notification may be sent to the service provider. The provider will then manually decide whether to participate or not. An appropriate message will be sent to the composite service based on the provider's decision.

Once faults are detected by coordinators, they need to be propagated to relevant composite services. In the rest of this chapter, we describe the algorithms executed by senders and receivers for propagating bottom-up faults. We assume that WS_i (with SS-S$_i$ as an attached sender) participates in CS_j (with SS-R$_j$ as attached receiver). We assume the existence of a pre-defined function *Agreed2Join()* used by services to decide whether they are willing to participate in compositions.Each service may provide its own definition and implementation of the *Agreed2Join()* function. We introduce two protocols: pure soft-state and soft-state with explicit removal. Communication messages in our protocol are shown in Figure 5.8.

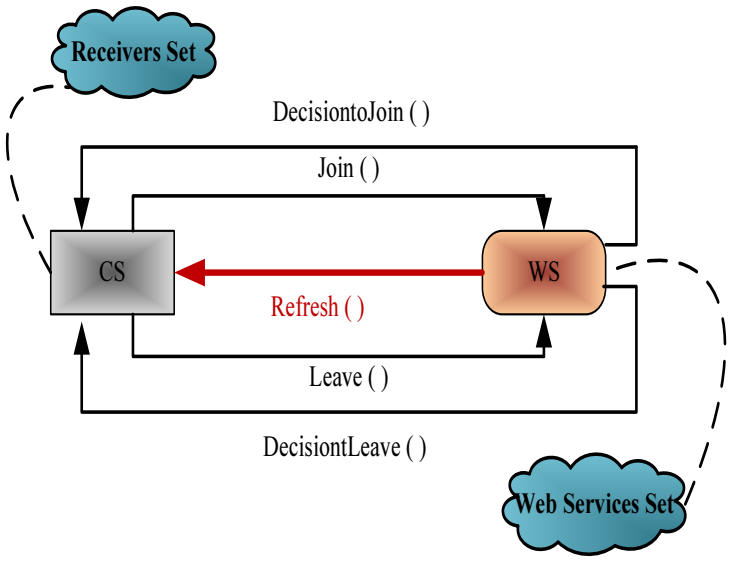


Figure 5.8: Communication between the composition service (CS) and Web service (WS).

5.3.1 Pure Soft-State

The Pure soft-state (Pure-SS) protocol adapts the well-know soft-state signaling described in (Ji, Ge, Kurose, and Towsley, Ji et al.2007; Garlan and Schmerl, Garlan and Schmerl2002) to service-oriented environments. It enables the propagation of participation refusal and policy changes faults to receivers. Physical faults are implicitly detected by receivers if they do not get Refresh() messages from a faulty sender during a certain period of time.

Sender's Algorithm

Figure 5.9 gives the algorithm executed by SS-S$_i$. SS-S$_i$ may receive two types of messages from SS-R$_j$: Join(SS-R$_j$) and Leave(). Join(SS-R$_j$) is the first message that SS-S$_i$ receives from SS-R$_j$; it invites WS_i to participate in CS_j (lines 1-13). SS-S$_i$ calls the *Agreed2Join()* function to figure out whether WS_i is willing to participate in CS_j. If *Agree2Join(SS-R$_j$)* returns False,

SS-S_i sends the *Decision2Join(SS-S_i,False)* message to SS-R_j. Otherwise, SS-S_i adds SS-R_j to its receivers. If SS-R_j is the first receiver of SS-S_i, SS-S_i initializes $State_i$ (we should initialize the value of $Receivers_i$ before line 04) and starts its t_{SSS} timer. Finally, SS-S_i sends its decision to SS-R_j through the *Decision2Join(SS-S_i,True)* message. At any time, SS-S_i may receive a Leave() message from SS-R_j (lines 14-16). This message indicates that CS_j is no longer using WS_i as a participant. In this case, SS-S_i removes SS-R_j from its receivers.

```
(01) At Reception of Join(SS-Rj) Do
(02)   If Agreed2Join(SS-Rj) = True  Then
(03)          Receiversi = Receiversi  {SS-Rj};
(04)          If Receiversi = 1 Then
(05)                  Statei.ChangeStatus = False;
(06)                  Statei.ChangeDetails = ;
(07)                  Start sss timer of SS-Si;
(08)          EndIf
(09)          Send Decision2Join(SS-Si,True) to SS-Rj
(10)   Else
(11)          Send Decision2Join(SS-Si,False) to SS-Rj
(12)   EndIf
(13) End

(14) At Reception of Leave(SS-Rj) Do

(15)    Receiversi = Receiversi  {SS-Rj};
(16) End

(17) At the occurrence of Change(C,S) in WSi Do
(18)   Statei.ChangeStatus = True;
(19)   Statei.ChangeDetails = Statei.ChangeDetails  {(C,S)};
(21) End

(22) At the end of sss timer of SS-Si Do
(23)    For each SS-Rj / SS-Rj  Receiversi Do
(24)       Send Refresh(Statei) to SS-Rj;
(25)    EndFor
(26)    Statei.ChangeStatus = False;
(27)    Statei.ChangeDetails = ;
(28)    Re-start sss timer of SS-Si;
(29) End
```

Figure 5.9: SS-S_i Sender Protocol for Pure-SS.

WS_i detects policy changes that may occur in the attached SS-S_i using one of the techniques described earlier. At the occurrence of a policy change (with a category C and scope S) in WS_i (lines 17-21), SS-S_i sets $State_i$.ChangeStatus to True. SS-S_i keeps track of that change by insert-

ing (C,S) in $State_i$.ChangeDetails. In this way, the state of SS-S_i to be sent to receivers at the end of t_{SSS} cycle includes all changes that have occurred during that cycle. At the end of each period (denoted by t_{SSS} timer), SS-S_i sends a Refresh() message to each one of its receivers (lines 22-29). This message includes $State_i$ as a parameter, hence notifying SS-R_j about all policy changes that occurred in WS_i during the last t_{SSS} period. SS-S_i then reinitializes $State_i$ and restarts its t_{SSS} timer.

Receiver's Algorithm

The aim of SS-R_j protocol is to detect faults in its senders. For that purpose, SS-R_j maintains a local table called *SR-Table*$_j$. SR-Table$_j$ allows SS-R_j to keep track of Refresh() messages transmitted by senders. It contains an entry for each SS-S_i that belongs to Sender(SS-R_j). Each entry contains two columns:

- *Refreshed: SR-Table*$_j$*[SS-S*$_i$*,Refreshed]* equals True iff SS-R_j received a Refresh() from SS-S_i in the current t_{SSR} cycle.
- *Retry: SR-Table*$_j$*[SS-S*$_i$*,Retry]* contains the number of consecutive cycles during which SS-R_j did not receive Refresh() from SS-S_i.

A temporary node failure in SS-S_i may prevent SS-S_i from sending Refresh() to SS-R_j during a t_{SSR} cycle. In this case, SS-R_j may want to give SS-S_i a second chance for sending Refresh() during the next t_{SSR} cycle. For that purpose, SS-R_j maintains a variable (positive integer) *Max-Retry*$_j$. If SS-R_j does not receive Refresh() from SS-S_i during Max-Retry$_j$ consecutive t_{SSR} cycles, it considers WS_i as faulty. The value of Max-Retry$_j$ is set by CS_j composer and may vary from a

composite service to another.

```
(01) At addition of WSi to CSj Do
(02)    Send Join(SS-Rj) to SS-Si;
(03) End

(04) At deletion of WSi from CSj Do
(05)    Send Leave(SS-Rj) to SS-Si;
(06)    Delete SS-Si entry from SR-Tablej;
(07) End

(08) At Reception of Decision2Join(SS-Si,decision) Do
(09)   If decision = True Then
(10)      Sendersj = Sendersj  {SS-Si};
(11)      Create an entry for SS-Si in SR-Tablej;
(12)      SR-Tablej[SS-Si,Refreshed] = False;
(13)      SR-Tablej[SS-Si,Retry] = 0;
(14)      If Sendersj = 1 Then
(15)         Start SSR timer of SS-Rj;
(16)      EndIf
(17)   Else React("Refusal",SS-Si);
(18)   EndIf
(19) End

(20) At Reception of Refresh(Statei) From SS-Si Do
(21)   SR-Tablej[SS-Si, Refreshed] = True;
(22)   If Statei.ChangeStatus = True  Then
(23)    React("changes",SS-Si, Statei.ChangeDetails);
(24)   EndIf
(25) End

(26) At the end of SSR timer of SS-Rj Do
(27)  For each SS-Si / SS-Si  Sendersj Do
(28)   If SR-Tablej[SS-Si, Refreshed] = True Then
(29)      SR-Tablej[SS-Si, Refreshed] = False;
(30)      SR-Tablej[SS-Si, Retry] = 0;
(31)   Else
(32)     SR-Tablej[SS-Si, Retry]++
(33)     If SR-Tablej[SS-Si,Retry] = Max-Retryj Then
(34)        React("No Refresh", SS-Si);
(35)     EndIf
(36)   EndIf
(37)  EndFor
(38)  Re-start SSR timer of SS-Rj;
(39) End
```

Figure 5.10: SS-R$_j$ Receiver Protocol for Pure-SS.

The smaller is Max-Retry$_j$, the more pessimistic is CS_j composer about the occurrence of faults in participants. SS-R$_j$ submits two types of messages to SS-S$_i$: Join() and Leave(). It also receives two types of messages from SS-S$_i$: Decision2Join() and Refresh(). Figure 5.10 gives the algorithm executed by SS-R$_j$. Whenever a new participant WS_i is added to CS_j, SS-R$_j$ sends a Join(SS-R$_j$) message to SS-S$_i$ (lines 1-3). At the deletion of WS_i from CS_j, SS-R$_j$ sends

a Leave(SS-R$_j$) message to SS-S$_i$ and removes SS-S$_i$ entry from SR-Table$_j$ (lines 4-7). At the reception of Decision2Join(SS-S$_i$,True), SS-R$_j$ adds SS-S$_i$ to the list of senders (lines 8-19). It also creates a new entry for SS-S$_i$ in SR-Table$_j$ and initializes the Refreshed and Retry columns of that entry to False and 0, respectively. If SS-S$_i$ is the first sender of SS-R$_j$, SS-R$_j$ starts its t_{SSR} timer. At the reception of Decision2Join(SS-S$_i$,False), SS-R$_j$ calls the *React()* function to process the participation refusal fault issued by SS-S$_i$.

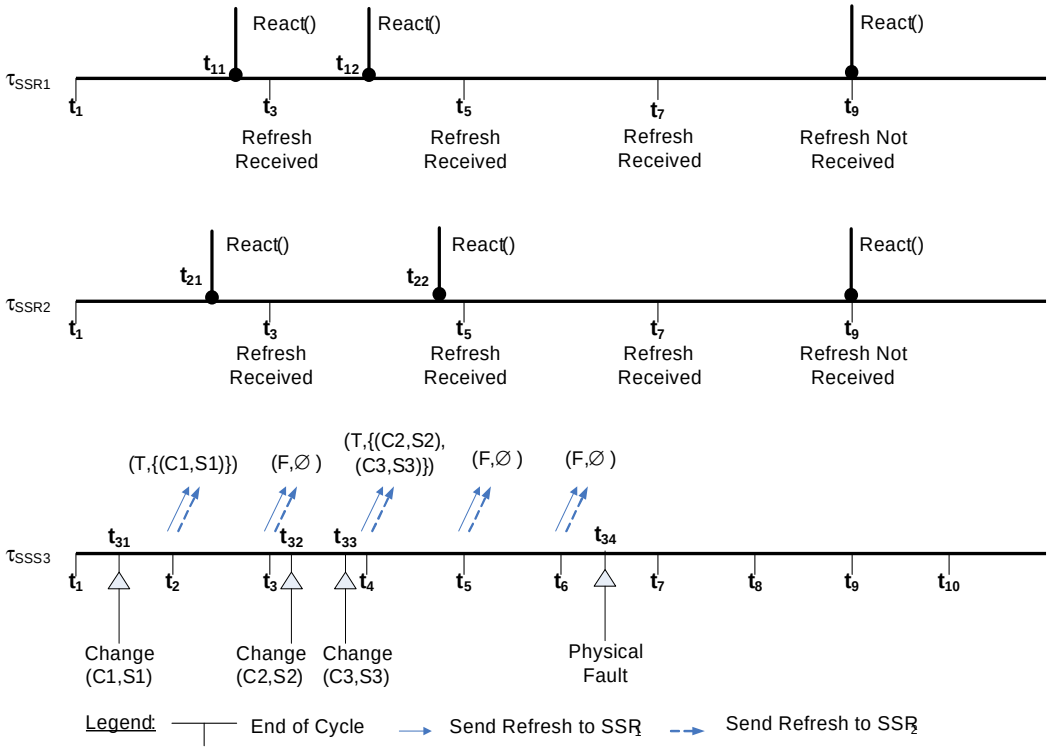


Figure 5.11: Pure-SS Protocol - Example.

At the reception of Refresh($State_i$), SS-R$_j$ sets SR-Table$_j$[SS-S$_i$,Refreshed] to True (lines 20-25). If $State_i$.ChangeStatus is True, SS-R$_j$ calls the *React()* function to process all changes that occurred in SS-S$_i$ during the last t_{SSR} cycle. At the end of t_{SSR} timer (lines 26-39), SS-R$_j$ checks

if it received Refresh() from each of its senders. If SS-R_j received Refresh() from SS-S_i, it re-initializes the Refreshed and Retry columns of SS-S_i entry in SR-Table$_j$ to False and 0, respectively. Otherwise, SS-R_j increments SR-Table$_j$[SS-R_j,Retry]. If SR-Table$_j$[SS-S_i,Retry] equals Max-Retry$_j$ (i.e., SS-R_j did not receive Refresh() from SS-S_i during Max-Retry$_j$ consecutive t_{SSR} cycles), SS-R_j assumes a physical (node) fault in SS-S_i and hence, calls the *React()* function to process that fault. SS-R_j finally restarts its t_{SSR} timer.

Let us consider the senders and receivers corresponding to our reference scenario. We focus on the Refresh() messages sent by SS-S_3 to SS-R_1 and SS-R_2. We assume that $t_{SSR1} = t_{SSR2} = 2 \times t_{SSS3}$. Figure 5.11 depicts the interactions between SS-S_3 and SS-R_1/SS-R_2. At time t_{31}, SS-S_3 detects a change (with category C_1 and scope S_1) in WS_3. SS-S_3 assigns True to State$_3$.ChangeStatus and inserts (C_1,S_1) in State$_3$.ChangeDetails. At time t_2, SS-S_3 sends Refresh(True,(C_1,S_1)) to SS-R_1 and SS-R_2, and re-initializes State$_3$. SS-R_1 and SS-R_2 process those changes by calling their *React()* function at times t_{11} and t_{21}, respectively. At t_3, SS-R_1 and SS-R_2 note the reception of the Refresh() sent by SS-S_3. At this same time, SS-S_3 sends Refresh() to both receivers with the parameters (False, $\emptyset$) since no changes have been detected in the second SS-S_3 cycle. SS-S_3 detects two changes (C_2,S_2) and (C_3,S_3) in WS_3 at t_{32} and t_{33}, respectively. At t_{33}, State$_3$.ChangeStatus equals True and State$_3$.ChangeDetails contains (C_2,S_2),(C_3,S_3). At t_4, SS-S_3 sends Refresh(True,(C_2,S_2),(C_3,S_3)) to SS-R_1 and SS-R_2. SS-R_1 and SS-R_2 process those changes at t_{12} and t_{22}, respectively. At t_5, SS-R_1 and SS-R_2 note the reception of the Refresh() sent by SS-S_3. At times t_5 and t_6, SS-S_3 sends Refresh() to SS-R_1 and SS-R_2 with the parameters (False,$\emptyset$) since no changes have been detected in the corresponding SS-S_3 cycle. At t_7, SS-R_1 and SS-R_2

note the reception of the Refresh() sent by SS-S_3. Let us now assume a server failure in WS_3 (and hence SS-S_3) at t_{34}. At t_9, SS-R_1 and SS-R_2 find out that they did not receive Refresh() from SS-S_3 during the last SS-R_2 cycle. If Max-Retry$_2$ is equal to 1, SS-R_1 and SS-R_2 conclude that SS-S_3 failed and hence call the *React()* function.

5.3.2 Soft-State with Explicit Removal

In the Pure-SS protocol, SS-R_j assumes a failure in SS-S_i if it does not receive Refresh() messages from SS-S_i after Max-Retry$_i$ SS-R_j cycles. This could happen because of a node fault (physical fault) or status change (logical fault) in SS-S_i. While physical faults are out of SS-S_i's control, status changes are scheduled by service providers and hence, can explicitly be related by SS-S_i to SS-R_j. This would have two major advantages. First, SS-R_j will be able to status change faults as soon as they occur in SS-S_i, instead of waiting the end of Max-Retry$_i$ SS-R_j cycles. Second, SS-R_j may differentiate between logical and physical faults and hence, react to them appropriately. The soft-state protocol with explicit removal (Removal-SS) extends Pure-SS with explicit removal messages; these messages announce future status change faults in the participants.

Sender's Algorithm

Figure 5.12 shows the steps of the Removal-SS algorithm executed by SS-S_i. The statements in lines 1-29 are the same as in Pure-SS. In what follows, we focus on the parts that are specific to Removal-SS (lines 30-44). If WS_i provider is scheduling a service shut-down (lines 30-34), SS-S_i sends an explicit Shutdown() message along with the down and up times (from and to, respectively) to each SS-R_j in Receivers(SS-S_i). In this case, SS-S_i will not send Refresh() messages during the

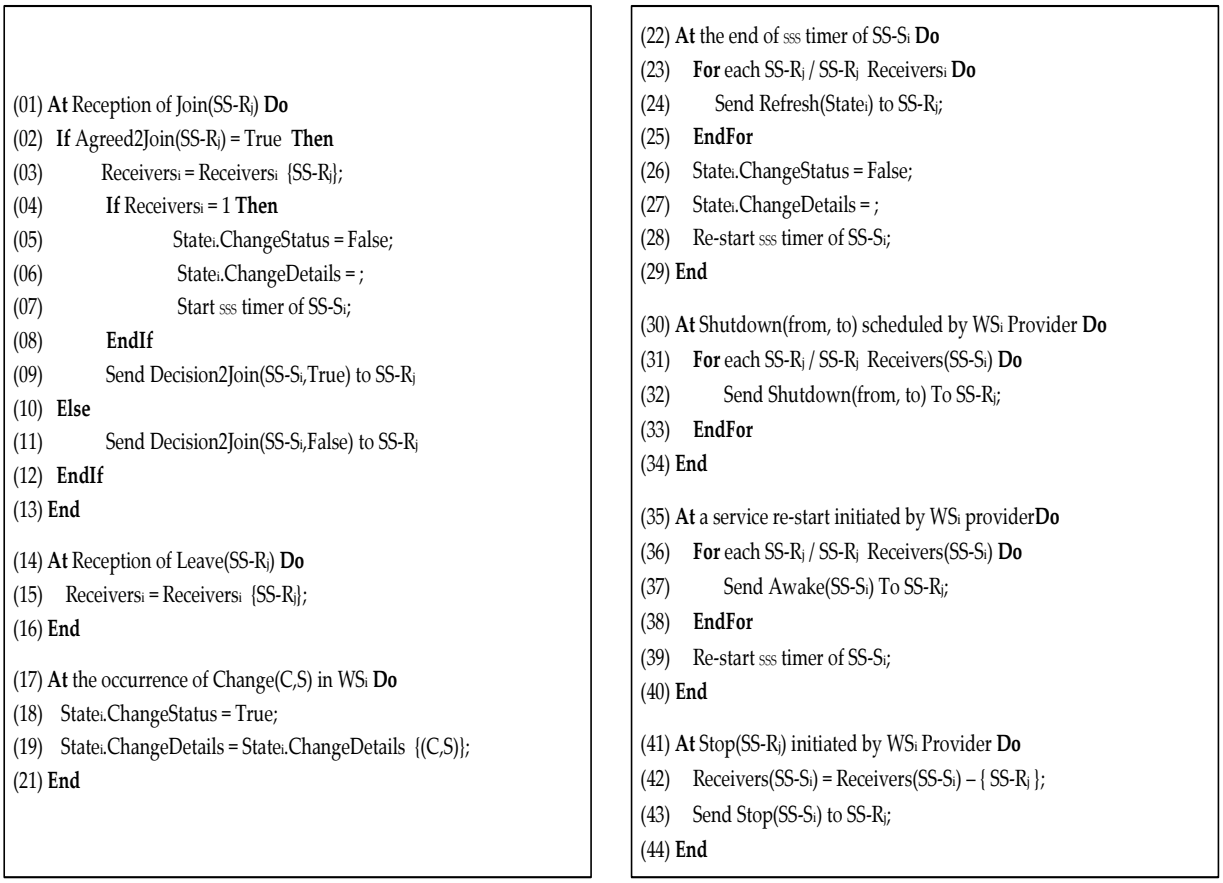


Figure 5.12: SS-S$_i$ Sender Protocol for Removal-SS.

period [from,to[. If WS_i provider decides to re-start WS_i (lines 35-40), SS-S$_i$ sends Awake() messages to all receivers in Receivers(SS-S$_i$) and re-starts its t_{SSS} timer. If WS_i provider decides to stop its service with CS_j (lines 41-44), SS-S$_i$ removes SS-R$_j$ from its receivers and sends an explicit removal message Stop() to SS-R$_j$. SS-S$_i$ will no longer send Refresh() messages to SS-R$_j$.

Receiver's Algorithm

Figure 5.13 describes the algorithm executed by SS-R$_i$. The statements in lines 1-25 are similar to the ones given in Pure-SS receiver's algorithm. To handle freeze faults (i.e., tempo-

rary shutdowns), SR-Table$_i$ entries are extended with "from" and "to" attributes. SR-Table$_i$[SS-S$_j$,from] and SR-Table$_i$[SS-S$_j$,to] contain the down and up times of the shutdown scheduled by WS_j provider, respectively. At the creation of an entry for SS-S$_j$ in SR-Table$_i$, SR-Table$_i$[SS-S$_j$,from] and SR-Table$_i$[SS-S$_j$,to] are initialized to 0 (lines 14-15). At the end of t_{SSR} timer (lines 20-25), SS-R$_i$ checks if it received a Refresh() message from senders. If a sender SS-S$_j$ is in the frozen status (lines 28-30), SS-R$_i$ does not expect to receive a Refresh() from SS-S$_j$ and hence skips SS-S$_j$.

Otherwise, SS-R$_i$ handles SS-S$_j$ as in the case of Pure-SS protocol (lines 31-41). At the reception of a Shutdown() message from SS-S$_j$ (lines 42-46), SS-R$_i$ processes the shutdown scheduled by SS-S$_j$ by calling the utility procedure Process-Shutdown(). At the reception of a Awake() message from SS-S$_j$ (lines 47-53), SS-R$_i$ checks if SS-S$_j$ still belongs to Senders(SS-R$_j$). SS-S$_j$ could have been removed from Senders(SS-R$_j$) as part of the Process-Shutdown() utility procedure. If SS-S$_j$ $\notin$ Senders(SS-R$_j$), SS-R$_i$ sends a Leave() message to SS-S$_j$. Otherwise, SS-R$_i$ re-initializes SR-Table$_i$[SS-S$_j$,DownTime] and SR-Table$_i$[SS-S$_j$,UpTime] with 0. At the reception of a Stop() message from SS-S$_j$ (lines 54-58), SS-R$_i$ deletes SS-S$_j$ entry in SR-Table and removes SS-S$_j$ from Senders(SS-R$_i$). Finally, it processes the stop notified by SS-S$_j$ by calling the utility procedure Process-Stop().

Let us consider our running scenario shown in Figure 5.14. We assume that at time t$_{31}$, WS_3 provider schedules a shutdown during the period [t$_{31}$,t$_5$]. This explicit removal is communicated to SS-R$_1$ and SS-R$_2$ via Shutdown() messages. SS-R$_1$ and SS-R$_2$ receive those messages at times t$_{11}$ and t$_{21}$, respectively, and call Process-Shutdown() procedure. Hence, they do not expect to receive

```
(01) At addition of WSi to CSj Do
(02)    Send Join(SS-Rj) to SS-Si;
(03) End

(04) At deletion of WSi from CSj Do
(05)    Send Leave(SS-Rj) to SS-Si;
(06)    Delete SS-Si entry from SR-Tablej;
(07) End

(08) At Reception of Decision2Join(SS-Si,decision) Do
(09)   If decision = True Then
(10)      Sendersj = Sendersj {SS-Si};
(11)      Create an entry for SS-Si in SR-Tablej;
(12)      SR-Tablej[SS-Si,Refreshed] = False;
(13)      SR-Tablej[SS-Si,Retry] = 0;
(14)      If Sendersj = 1 Then
(15)         Start SSR timer of SS-Rj;
(16)      EndIf
(17)   Else React("Refusal",SS-Si);
(18)   EndIf
(19) End

(20) At Reception of Refresh(Statei) From SS-Si Do
(21)   SR-Tablej[SS-Si, Refreshed] = True;
(22)   If Statei.ChangeStatus = True  Then
(23)    React("changes",SS-Si, Statei.ChangeDetails);
(24)   EndIf
(25) End

(26) At the end of SSR timer of SS-Ri Do
(27)  For each SS-Sj / SS-Sj  Senders(SS-Ri) Do
(28)   If time  SR-Tablei[SS-Sj,from]  time < SR-Tablei[SS-Sj,to]
(29)      Then Continue;
(30)   EndIf
(31)   If SR-Tablei[SS-Sj, Refreshed] = True
(32)         Then SR-Tablei[SS-Sj, Refreshed] = False;
(33)              SR-Tablei[SS-Sj, Retry] = 0;
(34)         Else SR-Tablei[SS-Sj, Retry
(35)              If SR-Tablei[SS-Sj, Retry] = Max-Retryi
(36)              Then React("No Refresh", SS-Si);
(37)              EndIf
(38)   EndIf
(39)  EndFor
(40)   Re-start SSR timer of SS-Ri;
(41) End
```

```
(42) At Reception of Shutdown(from, to) From SS-Sj Do
(43)   SR-Tablei[SS-Sj,from] = from;
(44)   SR-Tablei[SS-Sj,to] = to;
(45)   React("shutdown", SS-Sj, from, to);
(46) End

(47) At Reception of Awake(SS-Sj) From SS-Sj Do
(48)    If SS-Sj  Senders(SS-Ri)
(49)     Then Send Leave(SS-Ri) to SS-Sj;
(50)     Else SR-Tablei[SS-Sj,from] = 0;
(51)          SR-Tablei[SS-Sj,to] = 0;
(52)    EndIf
(53) End

(54) At Reception of Stop(SS-Sj)  From SS-Sj Do
(55)   Remove SS-Sj Entry From SR-Tablei;
(56)   Senders(SS-Ri) = Senders(SS-Ri) – { SS-Sj };
(57)   React("stop", SS-Sj);
(58) End
```

Figure 5.13: SS-R$_j$ Receiver Protocol for Removal-SS.

Refresh() messages at t_5. At time t_5, WS_3 is reinstated; SS-S_3 sends Awake() message to SS-R_1 and SS-R_2. SS-S_3 resumes sending Refresh() at time t_6. Assume that at time t_{32}, WS_3 provider would like to stop its service with SS-R_2. For that purpose, SS-S_3 sends a Stop() message to SS-R_2. SS-R_2 receives this message at time t_{22}, deletes SS-S_3 entry in SR-Table$_3$, removes SS-S_3 from Senders(SS-R_2), and calls Process-Stop() procedure. At times t_7 and t_9, SS-R_1 notes the reception of Refresh() messages. However, SS-R_2 does not expect the reception of such messages since WS_3

is no longer a composition participant.

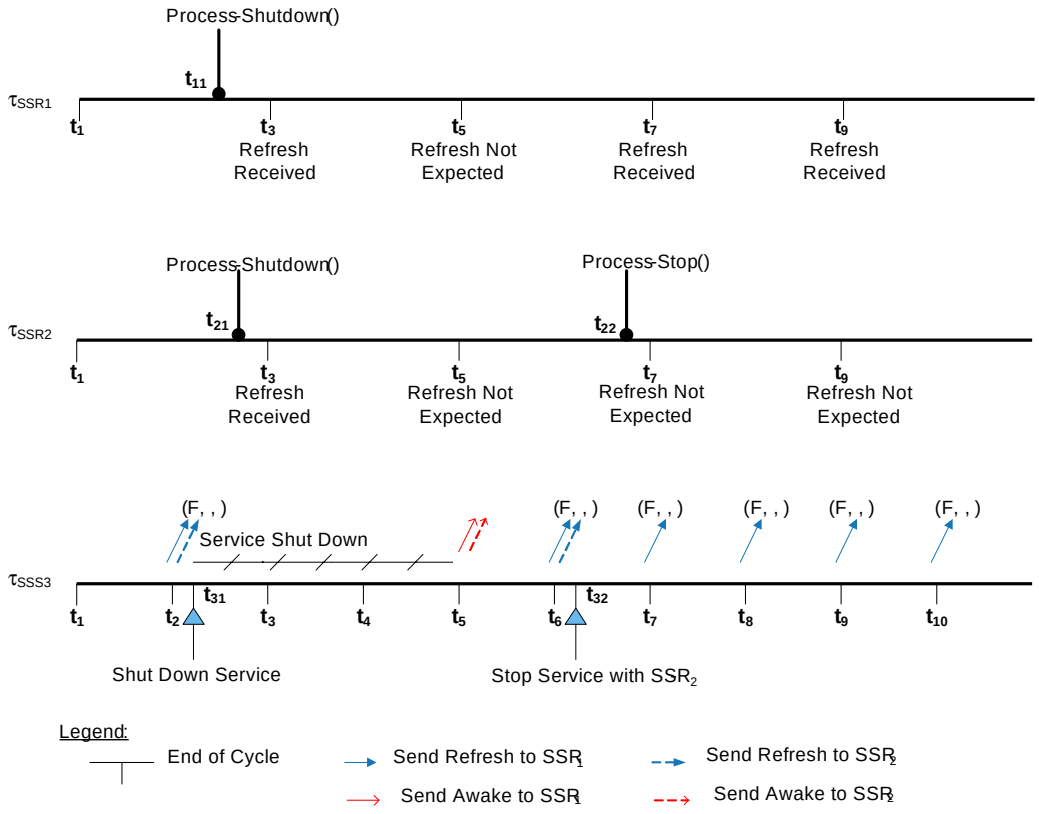


Figure 5.14: Removal-SS Protocol - Example.

5.4 Fault Reaction

Once the (composite) system determines that a fault has occurred, the next step is to prepare an optimal fault-response and recovery strategy. In this section, we first present Event-Condition-Action (ECA) rules, and then discuss the main recovery polices used by our framework. One of the issues to be considered is the conversion and negotiation of new policies for the replaced/composed web service. In our framework we propose an approach that uses ontologies and semantic web rules to ensure that policy information is correctly translated. The idea is to standardize all the

policies using a standard policy language i.e WS-Policy that could explicitly describe all the constraints i.e. privacy parameters that need to considered in a composition. Before introducing a new service to the composition, the systems checks for policy conformance of the selected service. We use ontologies to translate domain specific terms and use semantic web rules to convert compatible rules i.e. (uptime of 99.9% to downtime of 0.01% etc). Then we use our automated negotiation framework to negotiate a suitable solution for the composition (Hashmi, Alhosban, Malik, and Medjahed, Hashmi et al.2011).

Event processing is a method of tracking and analyzing streams of data about a specific event, and then deriving a conclusion based on these events. Complex Event Processing (CEP), is event processing that combines data from multiple sources to infer patterns in more complicated scenarios. The main goal of CEP is to identify meaningful events and response to them as quickly as possible. CEP employs different techniques such as detection of the complex pattern, event correlation, event abstraction, event hierarchies and relationship between events (Pascalau and Giurca, Pascalau and Giurca2009; Wasserkrug, Gal, Etzion, and Turchin, Wasserkrug et al.2008; Mozafari, Zeng, and Zaniolo, Mozafari et al.2012; Kellner and Fiege, Kellner and Fiege2009).

Event-driven systems are becoming the paradigm of choice for organizing many classes of loosely coupled and dynamic applications (Garlan and Schmerl, Garlan and Schmerl2002; Ghosh, Sharman, Raghav Rao, and Upadhyaya, Ghosh et al.2007; Verma and Sheth, Verma and Sheth2005; Guinea, Guinea2005; Kokash, Kokash2007; Brambilla, Ceri, Comai, and Tziviskou, Brambilla et al.2005). Events are typically used to provide users or systems awareness about specific sit-

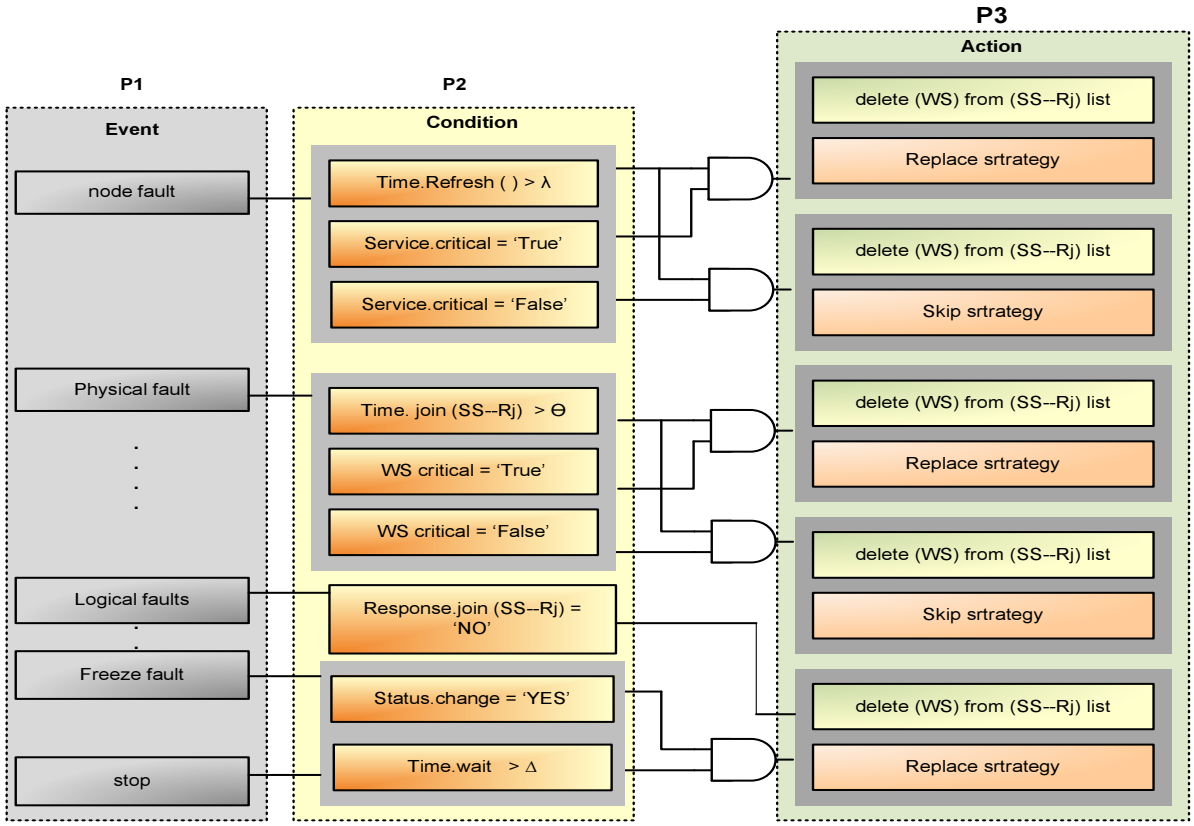


Figure 5.15: ECA Architecture.

uations that may occur during a system's execution span. System faults fall into one such category of notifications. To capture and react to events, each service also includes a set of rules. In this chapter, we adopt the ECA rules model (Garlan and Schmerl, Garlan and Schmerl2002) for defining these rules. Our selection is based on the fact that ECA rules specify constraints on the service properties and method and specify the reaction to requests and their responses. In addition, implementing intelligent systems require reasoning possibilities thus using CEP is not enough, so we use ECA to represent knowledge and act to the event individually. ECA rules automatically perform actions in response to events provided that slated condition hold. ECA rules

have a high level, declarative syntax and are amenable to analysis optimization techniques (Bailey, Poulovassilis, and Wood, Bailey et al.2002; Qiao, Zhong, Wang, and Li, Qiao et al.2007; Papamarkos, Poulovassilis, and Wood, Papamarkos et al.2011; Poulovassilis, Papamarkos, and Wood, Poulovassilis et al.2006).

We use the previously mentioned communication messages framework to support the ECA rules. Each ECA rule contains an event, condition, and action part. An event is a method invocation, a service state transition (e.g., termination of a Web service operation), or a combination of events via logical operations (AND, OR, NOT). A condition is a Boolean expression over the service state. An action can be a method invocation, a notification or a group of actions to be sequentially or currently executed. In what follows, we focus on the recovery policy action.

In general, *Events* are typically used to provide awareness about specific situations such as notify users about the shipment of a product or tracking changes that may occur in services. We define an event as the occurrence of fault (i.e., physical and logical fault). The main idea of using ECA is to react to changes that may occur in the services (e.g. a Web service is no longer available or out of service). Whenever a change event occurs, information about the corresponding change is sent to the composition service and/or other services that rely on it. These services react to the notified changes using their own policies via local ECA rules. The event part of ECA rule refers to change notification. The actions part allows for the specification of change policies. Figure 5.15 shows the set of events that may occur during the communication among the individual Web services and their corresponding composition services, along with their conditions and actions (recovery policies).

Every event (fault in our case) has a correlated condition and/or parameter associated to it. Starting from the event node fault, if the Web service exceeds λ (i.e., condition = 'true') then the Web service is not available and the composition will delete it from the list of component services, where λ is the time that the service takes to send periodic refresh notifications to its receiver services. So, if the failing Web service has a high critical value, the suggested action would be to replace it with a similar service.

Definition (Critical Service). For each composition, critical service is the component service that has the highest out degree of data dependency in the composition.

Hence in a composition, a service that provides the maximum number of input parameters (as its output parameters) for the participants of that composition is considered a *Critical Service*. There may be more than one critical service in the same composition. If the critical service fails in providing its service then the dependent component services cannot be executed and the composition will fail. For each composition, we first define the critical services which will help us in determining the action(s) in ECA rules. The degree of criticality is thus based on the tasks undertaken by these services, where each task contains one or more operations. The main factors for service criticality ($Crit_S$) are: the operation priority value that is provided by the consumer ($Wuser_i$) and operation criticality (defined as the ratio of the operation's execution time to the total execution time of the service).

$$OpCritical(op_i, service_j) = \frac{W_{Op_{ij}} \times \sum_{k=1}^{n} R_k}{|W| + |D|} \tag{5.1}$$

Where $W_{Op_{ij}}$ is the weight of the operation i in service j which is calculated by dividing the execution time of the operation over the total service time, R_k is the reputation of the service, W is the number of component services in the composition, and D is the number of Web services that depend on Web service j. By depend we mean that the operations of one service use the output of the other service. After determining the criticality of the operation we calculate the criticality of the task, and the criticality of the overall service ($Crit_S$).

$$TaskCriticality(task_k) = \frac{\sum_{i=1}^{z} Wuser_i \times OpCritical(op_i, service_j)}{\sum_{i=1}^{z} Wuser_i} \tag{5.2}$$

where z is the number of operations in task$_k$, and $Wuser_i$ is the user's weight for op$_i$. Then,

$$Crit_S = \sum_{i=1}^{m} TaskCriticality(task_i) \tag{5.3}$$

where m is the number of tasks in the service$_j$. If $Crit_S > \beta$ (where β is a predefined threshold) then the service is considered as a critical one.

In Figure 5.16, a sample rule in pseudo-code form (R1) is provided for a critical service. In contrast, when the condition = 'false' (i.e. service is not critical), the composite service deletes the Web service from the list and skips the task (R2). In case of a physical fault (i.e., the composition service did not receive any response from the component Web service for the join request message) for a critical service, we implement the replace strategy. We use θ to determine the maximum response time allowed to receive a reply for join request message (see R3 and R4 in Figure 5.16). In case of a logical fault, e.g., a freeze fault the providers shut down their services for a limited time period. In the stop fault, providers make their services permanently unavailable. In these two

cases, the action could be to retry the service after a certain time period (known as stop or freeze time). Hence, we defined Δ which is the freeze time as R5.

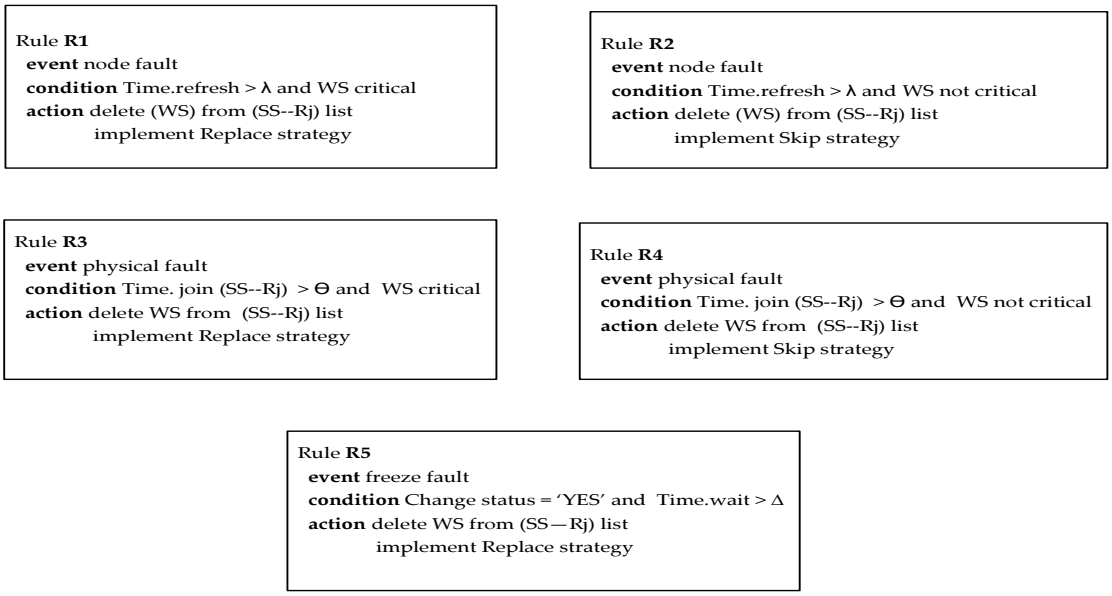


Figure 5.16: ECA examples.

There is another important concept for selecting the recovery policy which is Web Services Versioning. Because Web services are bound to change and evolve over time and the loose coupling principles of SOA imply that service providers can release a new version of a shared service without waiting for consumers to adapt, and that service consumers should test and certify on a new shared service version before switching. Consequently, you might need to have multiple versions of a shared service running concurrently and simultaneously accessible by different service consumers. Some service consumers might need to continue using an old version of a service until migration of the consumer code occurs. Therefore, Web services versioning is an important subject that should be considered carefully in all enterprise SOA approaches (Ibrahim, Ibrahim2009;

Frank, Lam, Fong, Fang, and Khangaonkar, Frank et al.2008). Service versioning comprises service specifications as observed at discrete points in time. These are identifiable by a version identification number; each version is agnostic of the others and managed individually. Each of the service versions is created by applying a number of changes to a previous service version, which can be thought of as the baseline for that version. Information regarding the baseline of each version, and how a service version differs from its baseline constitutes the version history of a given service (Andrikopoulos, Benbernou, and Papazoglou, Andrikopoulos et al.2008).

CHAPTER 6

PERFORMANCE ANALYSIS

In this chapter, we present a comprehensive performance analysis of the different stages of our work. Each stage has its own experiments, analysis and results. Our prototype SURETY is deployed to study the performance of different fault management strategies. The WSDream-QoSDataset (Zheng and Lyu, Zheng and Lyu2010) is used, which contains 150 Web services distributed in computer nodes located all over the world (i.e., distributed in 22 different countries), where each Web service is invoked 100 times by a service user. Planet-Lab is employed for monitoring the Web services. The service users observe, collect, and contribute the failure data of the selected Web services to our server, which is implemented in C# using Asp.Net running on Microsoft .Net version 3.5 and SQL as the back-end database. The following table (Table 6.1) provides a sample of the experiment run.

Table 6.1: Sample Web services run using Planet-Lab

Client IP	Response time (ms)	Data size	Message
35.9.27.26	2736	582	OK
35.9.27.26	804	14419	OK
35.9.27.26	20176	2624	connect timed out

The notations used hereafter are listed in Table 6.2. Most of the terms in the table are self-explanatory.

Table 6.2: Definition of Symbols

Symbol	Definition
T	The total execution time.
t_0	Start time.
t_n	End time.
t_i	Time at which a new service is invoked.
k	Number of services.
$P(x)^t$	Fault occurrence likelihood for service$_x$ when invoked at time t.
λ_i	Weight of service$_i$ in relation to T.
λ'_i	Weight of service$_i$ in relation to $(T - t_i)$.
Δ_i	First-hand fault history ratio of service$_i$.
Δ'_i	Second-hand fault history ratio of service$_i$.
$f(s_i)$	The priority of service$_i$ in the composition.

6.1 Fault Prediction

We simulated a services-based system complete with fault prediction, recovery strategies and performance measurement. The input to the system is an XML schema of the system that is used to exhibit the characteristics of a running system.

The experimental results based on the main scenario are discussed below for fault prediction, the experiment focus is on our in reducing the total execution time. The major motive is to not let the service executa if there is a high likelihood of it failing at run time (as this increases the total execution time). In Figure 6.1, we show a system with 11 services and 6 invocation points (details in these in Chapter 1). The invocation points are set at $t_1 = 30$ ms, $t_2 = 50$ ms, $t_3 = 450$ ms, $t_4 = 560$ ms, $t_5 = 670$ ms, $t_6 = 890$ ms with the total execution time (T) of 1000 ms . At t_1 the system invokes two services ($service_1$, $service_3$) in parallel, at t_2 the system uses probabilistic invocation for three services ($service_{3a}$, $service_{3b}$, $service_{3c}$). At t_3 the system invokes one service ($service_2$) which is sequential invocation, and at t_4 the invocation is synchronous for one service $service_4$.

At t_5 the invocation is again probabilistic for three services ($service_5$, $service_6$, $service_7$) and at t_6 the system invokes one service ($service_8$). Table 6.3 shows a sample (i.e. these are not constant) of the different parameter values for all 6 invocation points.

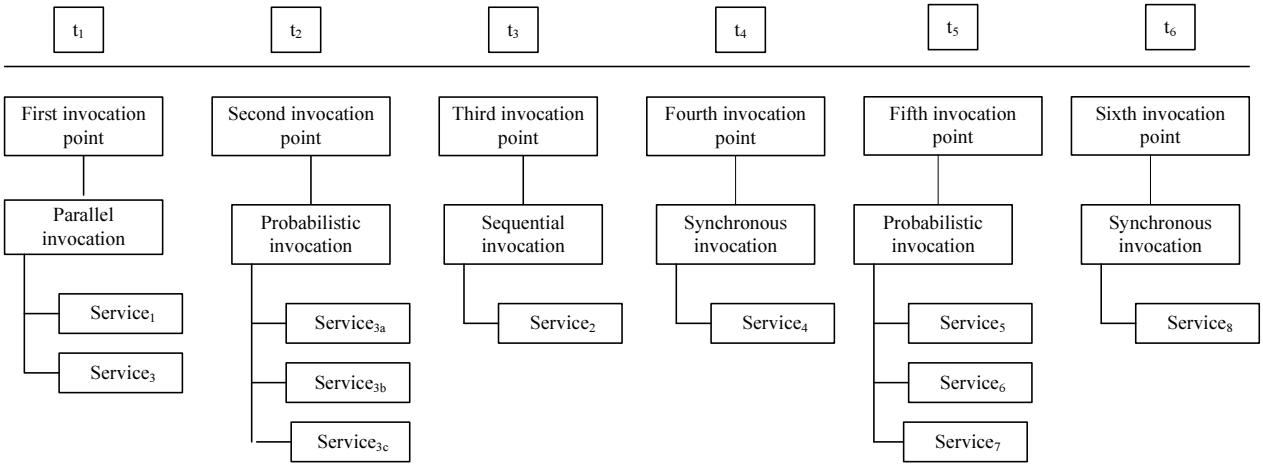


Figure 6.1: Simulation Environment of Eleven Services and Six Invocation Points.

Table 6.3: Service Parameters at Invocation Points

Invocation Point	Service	Time	Priority	λ_i	λ'_i	Δ_i	Δ'_i	$P(s_i)$
1	$Service_1$	180	60%	0.18	0.1856	0.30	0.15	0.2289
1	$Service_3$	250	40%	0.25	0.2577	0.40	0.60	0.2875
2	$Service_{3a}$	80	70%	0.08	0.8421	0.90	0.70	0.7498
2	$Service_{3b}$	90	50%	0.09	0.0947	0	0.20	0.1241
2	$Service_{3c}$	80	30%	0.08	0.0842	0.50	0.40	0.1120
3	$Service_2$	150	80%	0.15	0.2727	0.70	0.80	0.5414
4	$Service_4$	1000	80%	0.1	0.2273	0.60	0.80	0.4471
5	$Service_5$	80	90%	0.08	0.2424	0.70	0.65	0.2883
5	$Service_6$	70	80%	0.07	0.2121	0.50	0.80	0.3522
5	$Service_7$	80	90%	0.08	0.2424	0.75	0.90	0.6395
6	$Service_8$	100	80%	0.1	0.9091	0.70	0.80	0.0374

After experimenting multiple threshold values we assume the different theta values for this experiment i.e., $\theta_1 = 0.50$, $\theta_2 = 0.60$. The table lists the priority of each service involved, the services'

time weights and their history ratios (from internal and external experiences). Using Equation 3.5, FOLT calculated the fault likelihood at the first invocation point (Parallel invocation) to be Ppar =0.4505. Since $service_1$ and $service_3$ had a very low fault likelihood, this in turn implied that the invocation point fault likelihood was lower. In this case $P_{par} < \theta_1$, FOLT did not build any plan and continued with the system execution. For the second invocation point (Probabilistic invocation), as per the given parameters FOLT calculated the fault likelihood using Equation 9 to be P_{pro}=0.0104. Hence, the system did not build a recovery plan and continued its execution. Similarly at third invocation point (Sequential invocation): the fault likelihood was calculated using Equation 6 to be P_{seq}=0.5414. In this case $P_{seq} > \theta_1$, the system did build a recovery plan and continued its execution. However, the execution of the created plan had to wait until the occurrence of fault because $P_{seq} < \theta_2$. Fourth invocation point (Synchronous invocation): The fault likelihood was same as of $service_4$ = 0.4471. Fifth invocation point (Probabilistic invocation): The fault likelihood calculated by FOLT was P_{pro}=0.0649. Since $service_7$ had a high fault likelihood and the other two services had low fault likelihood , this in turn implied that the invocation point fault likelihood was lower. In the case that the selected service was $service_7$, the system will build a recovery plan and execute it(fault likelihood of $service_7 > \theta_2$). Sixth invocation point (Asynchronous invocation): The fault likelihood of this invocation point was same as that of $service_8$ = 0.0374. For this invocation point the system did not create any plan.

In Figure 6.2-(a) We can see the eleven services in this system and their fault likelihoods. We notice that $servie_{3a}$ has the highest fault likelihood and $service_8$ has the lowest fault likelihoods. These results are based on the different service's weight, history, behavior, invocation time and

priority. Figure 6.2-(b) shows the fault likelihood for each invocation point where the highest fault likelihood was at t_3 and the lowest fault likelihood was at t_2. Figure 6.2-(c) shows the relationship between the priority and the fault likelihood. For example, $service_{3a}$ has a priority of 70% and the fault likelihood is 0.7498, however, the priority for $service_7$ is 90% and the fault likelihood is 0.6395, because it has lower weight. Figure 6.2-(d) presents the relationship between service weight and the fault occurrence likelihood.

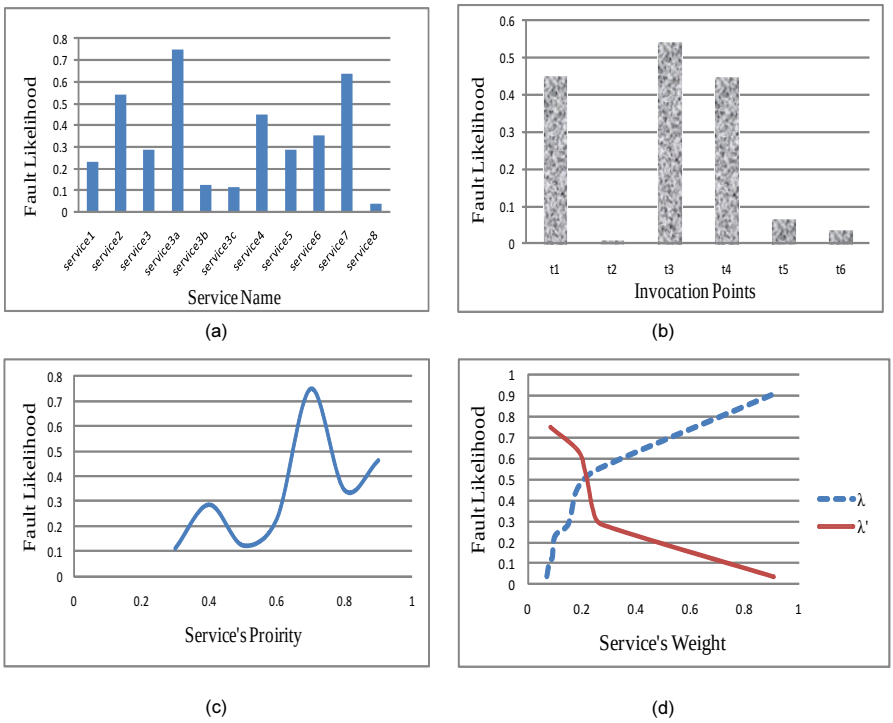
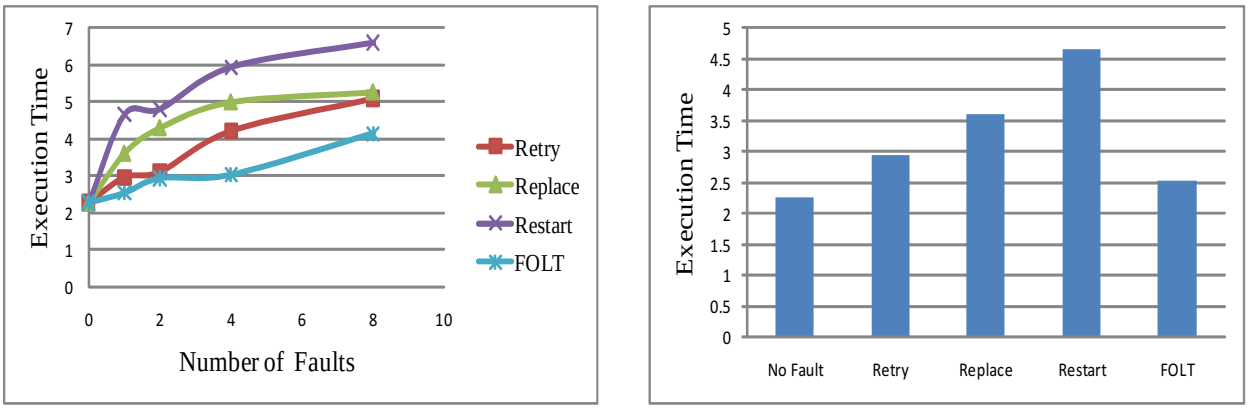


Figure 6.2: (a) Fault Likelihood (b) Invocations Points (c) Priority (d) Service Weight

We also performed experiments to assess the FOLT approach's efficiency. Figure 6.3-(a) shows the comparison between FOLT, no fault and systems that use replace, retry and restart as recovery techniques. Here total execution time is plotted on the y-axis and the number of faults on the x-axis. With increasing number of faults, the execution time also increases. However, FOLT takes



(a) (b)

Figure 6.3: Execution Time Comparisons

less time than compared techniques. This is due to the fact that FOLT preempts a fault and builds a recovery plan for it. Figure 6.3-(b) shows the total execution time comparisons for the five systems. Here we fix the number of faults to four.

6.2 Semantic Similarity

In this section, we define an analytical model to study the performance of the proposed semantic similarity technique (S^2R). Our Analytical model has 1000 Web services that are divided into three categories. We change the number of polices that are evaluated every time (e.g., 2, 4, 8 policies) while keeping other variables such as number of context specifications per policy, number of members per category, etc. fixed. We focus on computing the total time and search space complexity for checking the similarity degree of the target Web services through our three levels (for details see Chapter 2). We compare our technique with three similar existing works through this analytical model. Table 6.4 defines the parameters and symbols used here after. We assume

that Web services are divided into categories (e.g., under UDDI categories).

Table 6.4: Definition of Symbols

Symbol	Definition
X_j	The value of jth component of consumer's vector.
Y_{ij}	The value of jth component of ith Provider's vector.
$X_{j(min)}$	The minimum allowed value of jth component of consumer's vector as provided by the consumer.
$X_{j(max)}$	The maximum allowed value of jth component of consumer's vector as provided by the consumer.
$Y_{ij(min)}$	The minimum allowed value of jth component of ith Provider's vector as provided by the provider.
$Y_{ij(max)}$	The maximum allowed value of jth component of ith Provider's vector as provided by the provider.
WX_j	The weight of jth component of consumer's vector as provided by the consumer.
WY_{ij}	The weight of jth component of ith Provider's vector as provided by the provider.
r_j	Utility of the solution s for participant j.
R_s	Utility of the solution s (for all participants).
N_{com}	Number of categories.
N_p	Number of policies per service.
N_{cs}	Number of context specifications per policy.
N_{member}	Number of members per category.
N_r	Number of rules.
N_s	Number of services in a particular category.
T_{pall}	Time to fetch all policies of a service.
T_{pall_1}	The time to parse a service description and the network transmission delay.
T_{pall_2}	The time spent by each category to process its sub request.
T_{pone}	Time to fetch one policy of a service.
T_{cs}	Time to get a context specification.
T_{XML}	Time to parse a service description.
T_{Net}	Network transmission delay.
T_{rep}	Time spent to assess a reply from a category.

To simplify the analysis, we assume that the times to retrieve a description from a service registry and parse that description are fixed values. Based on the categories, we compute the

average matching time T_{match} which is the time it takes to find the similar services.

$$T_{match} = \frac{(T_{MINmatch} + T_{MAXmatch})}{2} \tag{6.1}$$

where $T_{MINmatch}$ is the best case matching time and $T_{MAXmatch}$ is the worst case matching time. T_{match} includes a polling time (T_{poll}) and a decision time (T_{dec}), where, T_{poll} is a combination of T_{poll1} and T_{poll2}. T_{poll1} includes the time it takes to fetch the policies, parse a service description and the network transmission delay. In the best case, the Web service would have only a single policy ($T_{MINpoll1}$) and in the worst case it may have N_{com} polices ($T_{MAXpoll1}$). Hence,

$$T_{MINpoll1} = T_{pone} + T_{XML} + T_{Net} \tag{6.2}$$

$$T_{MAXpoll1} = T_{pone} + N_{com} \times (T_{XML} + T_{Net}) \tag{6.3}$$

T_{poll2} includes the time spent in each category to process its sub request.

$$T_{MINpoll2} = 2 \times T_{pone} + 2 \times T_{cs} + 4 \times T_{XML} \tag{6.4}$$

We multiply T_{cs} by two because we need to compare each policy twice: once for the provider and once for the consumer. At a minimum, each policy would be compared to a single policy on both sides. Similarly, we multiply T_{XML} by four because we need to parse the description of XML four times (we need to parse XML files twice for the consumer and twice for the provider: once

for determining the properties and once for determining the policies).

$$T_{MAXpoll2} = T_{pone} + N_{cs} \times (T_{XML} + N_s \times (T_{pone} + T_{XML} + 2 \times Tcs + N_r \times (2 \times T_{XML}))) \quad (6.5)$$

In calculating $T_{MINpoll2}$ we multiply T_{pone} by two because we retrieve the policy for the source and the category member. The decision time (T_{dec}) includes the network delay and the time spent to asses a reply from the Web services under the same category. In the best case,

$$T_{MINdec} = T_{Net} + T_{rep} \quad (6.6)$$

$$T_{MAXdec} = N_{com} \times (T_{Net} + T_{rep}) \quad (6.7)$$

Based on the previous equations, T_{match} is then,

$$T_{match} = \frac{(T_{MINpoll1} + T_{MAXpoll1} + T_{MINpoll2} + N_{com} \times T_{MAXpoll2} + T_{MINdec} + T_{MAXdec})}{2} \quad (6.8)$$

The previous formulas give matching times for each technique. In what follows, we calculate the total matching time for the four techniques: S^2R, CME (Medjahed and Atif, Medjahed and Atif2007), CCB (Segev, Segev2008) and Brute-force (for details see Chapter 4).

Figure 6.4a. shows a comparison between the Brute-force method (exhaustive search), CME (Medjahed and Atif, Medjahed and Atif2007), CCB (Segev, Segev2008) and S^2R for service matching time based on the number of services. Note that Brute-force has the highest matching time especially when we have a large number of services. CME performs better than Brute-force method,

but still takes more time than CCB. However, our method provides the lowest matching time, even if the number of services is large.

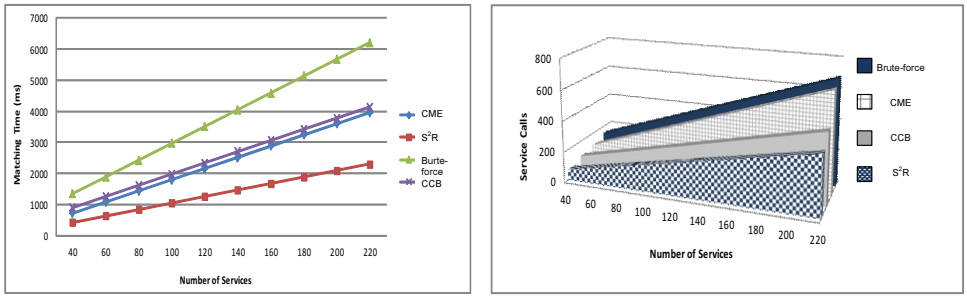


Figure 6.4: Matching Time and Search Space analysis.

Figure 6.4b. shows the relationship between the number of services and the search space for each matching method. We can see that Brute-force method has the largest search space and the smallest search is attributed to S^2R.

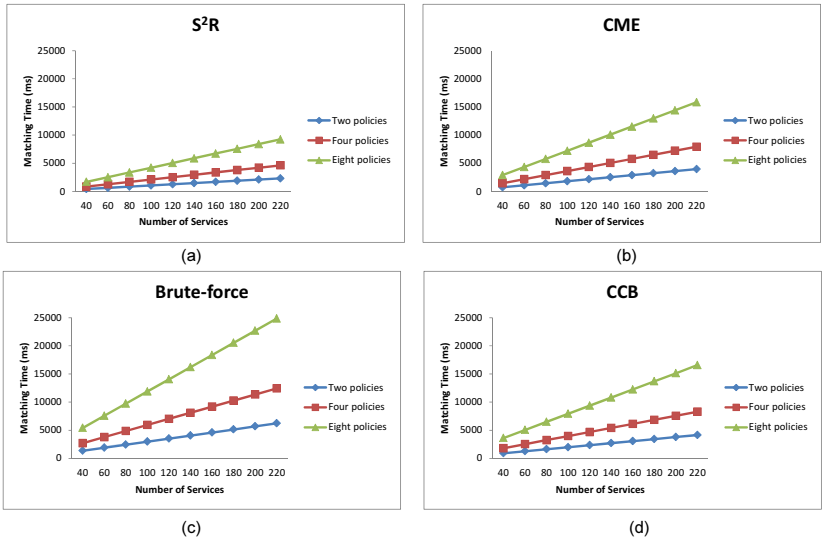


Figure 6.5: Scalability analysis for (a) S^2R. (b) CME. (c) Brute-force. (d) CBB.

In the last set of experiments, we evaluate the services matching time with variable number of

policies. Figure 6.4 presents the results of two, four, and eight policies. In Figure 6.5a. we can see that S^{2}R took a maximum matching time of (8,000 and 10,000 ms) when we use eight policies. However, in the Brute-force method (see Figure 6.5c.) the maximum time is between (22,000 and 25,000 ms). Figure 6.5b. shows that the maximum time using CME which is between (14,500 and 16,000 ms). Moreover, Figure 6.5d. shows that CCB took the maximum time of (15,000 to 17,000 ms). This shows that S^{2}R provides better matching time for all variable number of policies.

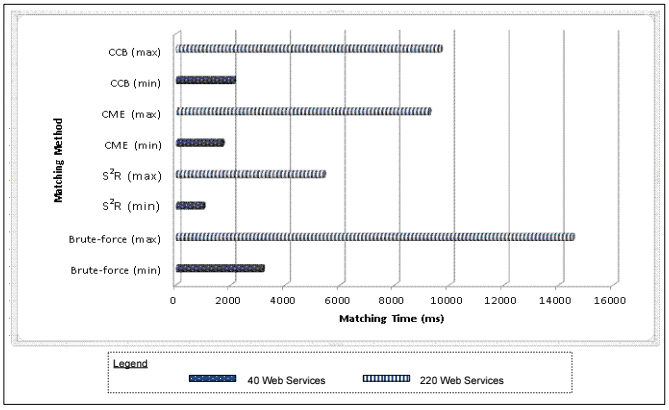


Figure 6.6: Maximum and minimum matching times.

Figure 6.6 shows the four techniques with their maximum and minimum matching times. We can see that S^{2}R takes the least amount of time to find the matching services, and scales very well when the number of Web services is increased (shown in Figure 6.6 from 40 to 220).

Figure 6.7 shows a comparison between the Brute-force method (exhaustive search), CME (Medjahed and Atif, Medjahed and Atif2007), CCB (Segev, Segev2008) and MASC for service matching time based on the number of services required by the consumer.

Note that Brute-force has the highest matching time especially when we have a large number of services. CME performs better than Brute-force method, but still takes more time than CCB.

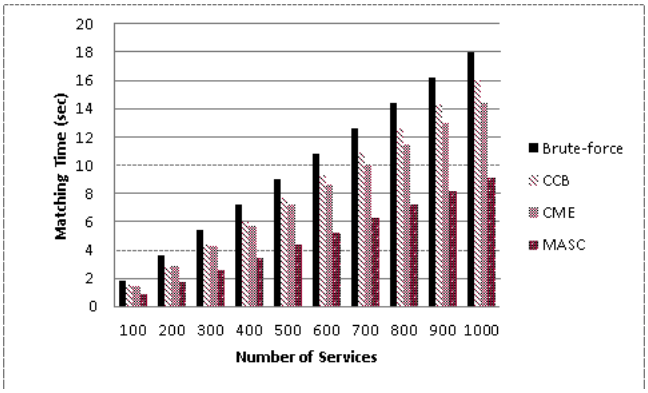


Figure 6.7: Service Matching Times.

However, our method provides the lowest matching time, even if the number of services is large because for each filter we reduced the number of compared services. Figure 6.8 shows the relationship between the number of services and the search space (number of service calls) for each matching method. We can see that our technique (MASC) perform better in comparison with other approaches because CME calls each service three times for comparison, CCB calls each service twice, while MASC calls just the candidate services twice.

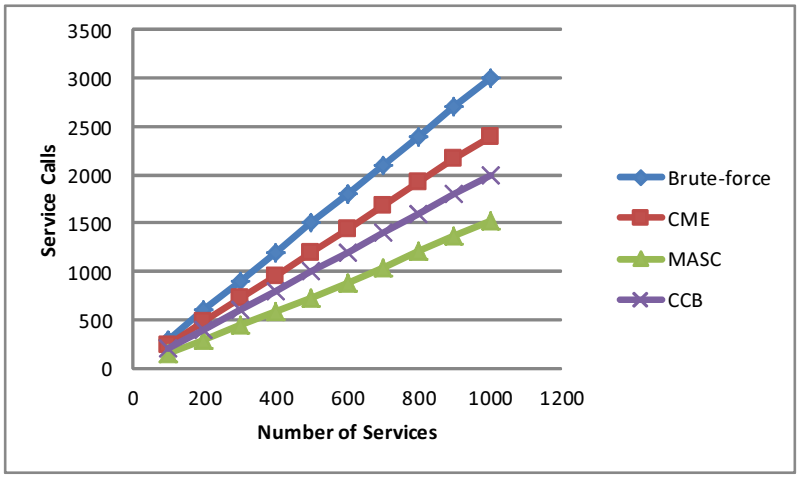


Figure 6.8: Service Calls made in relation to the number of services.

Figure 6.9 shows the different clusters of services obtained after the HMM and utility calculations. The HMM classifies the services into two classes (acceptable and unacceptable), while based on the utility values, the services are clustered separately. In the experiments, we have used WEKA for cluster construction.

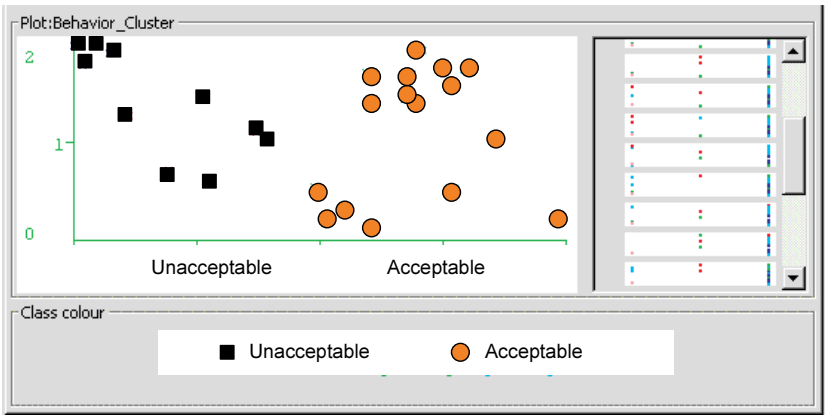


Figure 6.9: Behavioral patterns clustering using WEKA.

Moreover, we test different classification and clustering techniques to find the most appropriate technique for the behavior pattern classification. Table 6.5 shows the comparison between the classification techniques: 48, J48 with cross-validation, BFTree, NativeBayes and LADTree. The comparison is based on correctly classified instances, incorrectly classified instances,kappa statistic, mean absolute error, root mean squared error, relative absolute error and root relative squared error. The experiments show that NativeBayes performs better than other techniques for our algorithm and data set.

Table 6.5: Classification and clustering techniques

Technique	J48	BFTree	NativeBayes	LADTree
Correctly Classified	64%	68%	**98%**	96%
Incorrectly Classified	36%	32%	**2%**	4%
Mean absolute error	0.2036	0.1784	**0.0073**	0.0555
Mean squared error	0.3345	0.2986	**0.0457**	0.1248

6.3 Dynamic Planning

In this section, we use both analytical analysis and simulation to evaluate FLEX's performance. We focus on evaluating the overall system reliability by evaluating the reliability for each invocation point based on different recovery plans. We then compare our technique with four similar existing works. The QoS model formally defines a set of quality parameters for Web services. We define the best quality service through two sets: negative and positive QoS parameters. In the negative QoS parameters the higher value is the worst quality (e.g., the higher response time is the worse quality). In the positive QoS parameters the lower value is the worse quality (e.g., the lower reputation is the worse quality). We calculate the following QoS parameters:

Latency is measured as: $Latency(s_i) = Time_p + Time_r$. Where $Time_p$ is the time of processing the request and $Time_r$ is the transmission delay. Reliability is calculate as: $Reliability(s_i) = 1 - P(s_i)^t$. Availability is measured as: $Availability(s_i) = \frac{N_v}{T_v}$. Where N_v is the number of times that s_i was available and T_v is the total times that s_i was invoked.

Based on the above definitions, we use a score function which computes a scalar value from the (normalized) QoS parameters for assessing the service recovery plan (SRP):

$$Score(s_i) = (\sum_{negative} w_i \times \frac{QoS_i^{max} - QoS_i}{QoS_i^{max} - QoS_i^{min}} + \sum_{positive} w_i \times \frac{QoS_i - QoS_i^{min}}{QoS_i^{max} - QoS_i^{min}}) \quad (6.9)$$

where QoS_i^{max} is the maximum value for i^{th} QoS parameters and QoS_i^{min} is the minimum. Then reliability scores for the different planning strategies (details in Chapter 2) are computed as:

- Retry (s_i): find the score for s_i and multiply it by the same score:

$$Score_{retry} = (Score_s)^2 \quad (6.10)$$

- Retry-Until(s_i, ρ): find the score for s_i and take the power to the number of retries:

$$Score_{Retry-Until} = (Score_s)^m \quad (6.11)$$

- Replace(s_i, s_j): find the score for s_i and s_j:

$$Score_{Replace} = (Score_{si}) \times (Score_{sj}) + \partial \quad (6.12)$$

  where ∂ is the cost of fining similar service.

- Replicate($s_i, (s_1 ... s_k)$): find the score for the replicated services from s_i and s_k:

$$Score_{Replicate} = (Score_{si}) \times \frac{\sum_{h=1}^{k}(score_{sh})}{number\ of\ replicas} + \partial \quad (6.13)$$

The overall system reliability depends on the reliability for each invocation point, thus we use a linear function to approximate:

$$Total\ reliability = \sum_{\alpha=1}^{I} log(Reliability(\alpha)) \tag{6.14}$$

where I is the number of invocation points in the system and $Reliability(\alpha)$ is the reliability for the invocation point α.

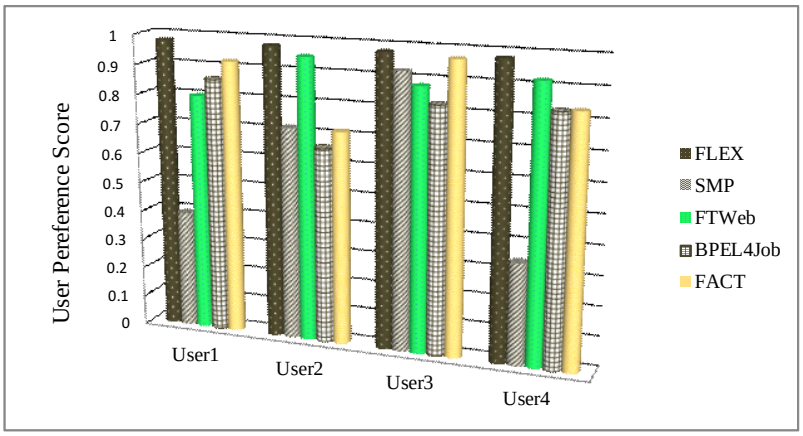


Figure 6.10: Assessed service score for user's requirements.

Figure 6.10 compares the scores for various techniques (i.e., FLEX, FACT (Liu, Li, Huang, and Xiao, Liu et al.2009), BPEL4Job (Tan, Fong, and Bobroff, Tan et al.2010), SMP (Dai, Yang, and Zhang, Dai et al.2009)and FTWeb (Santos, Lung, and Montez, Santos et al.2005)), for the different QoS requirements of four users. We can see that SMP is a good choice for $User_1$ and $User_3$ incase we had to replace Web service$_1$ with the best performing candidate Web service. Furthermore, FTWeb is the good choice for obtaining a good consistent score since it uses replication. BPEL4Job depends on the QoS values of a Web service and since Web service$_1$ has high QoS parameters values, it gives BPEL4Job a higher score. Note that FLEX performs better than the above

mentioned techniques even though we may have different preferences for user requirements, as we dynamically select the best recovery plan containing Web services with the highest score.

In FLEX, if we have more than one recovery plan at any invocation point, then we consider two factors for selecting the best recovery strategy. One is the type of service invocation point (i.e., sequential, parallel, etc) and second is the type of recovery plan (replace, retry, etc). We use a greedy approach)to select the most suitable recovery plan at each invocation point. This approach guarantees the selection of the best possible recovery plan (based on the cumulative score) at each invocation point and hence better system reliability. In addition to calculating the score of the recovery plans, FLEX calculates the reliability of each invocation point based on the Equations (4, 5, 6, 7, and 8). Table 6.6 summarizes the invocation points' reliability.

Table 6.6: The conditions for each planning strategy

Invocation	Reliability (R)
Sequential(S : A)	$R = R(S) \times R(A)$
Parallel (S : S_1, S_2)	$R = R(S) \times \sqcap_{i=1}^{2} R(S_i)$
Probabilistic (S : S_1—p, S_2—1-p)	$R = R(S) \times \sum_{i=1}^{n} p * R(S_i)$
Circular (S—m)	$R = R(S)^m$

Figure 6.11 shows the relationship between the service execution time and the failure ratio. Each sub-figure represents a category in our running example (e.g., hotel category, flight category, etc.,). We have 14 Web services under each category, we run each Web service 100 times and monitor the failure ratio and the execution time. We found that, when the service execution time increases the failure ratio also increases. In Figure 6.12, we show the results of running SURETY as a composite service with FLEX support. The same configuration of services is run with similar fault management techniques defined in the related work section, such as FACT (Liu, Li, Huang,

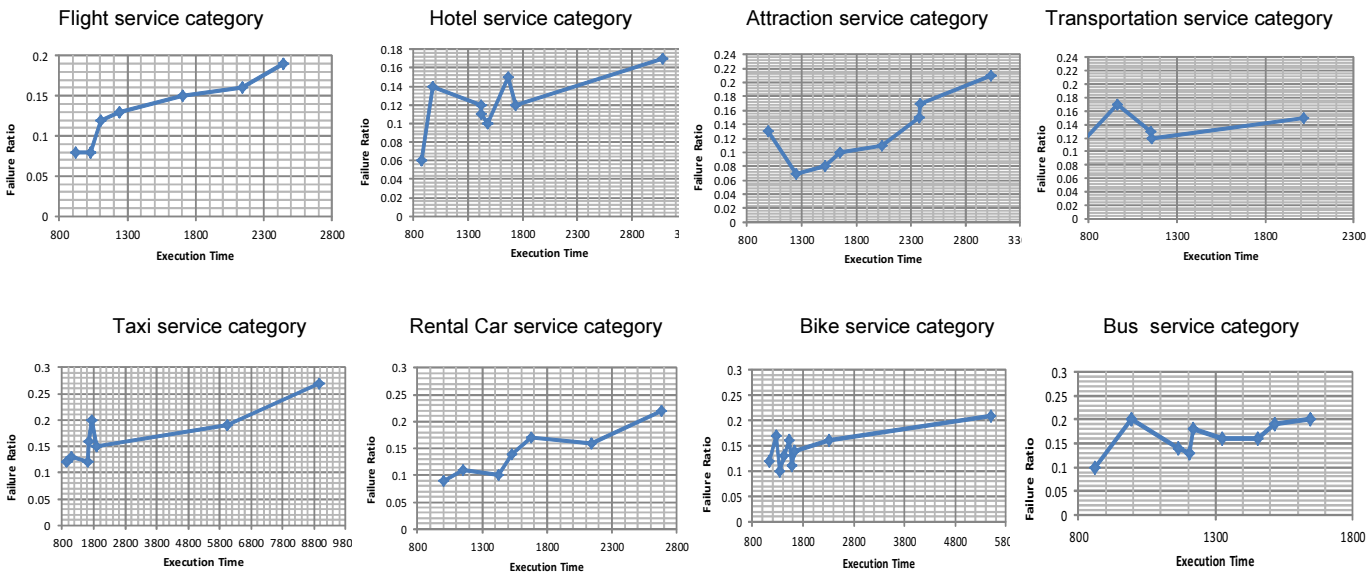


Figure 6.11: Impact of the services response time on the failure ratio.

and Xiao, Liu et al.2009), BPEL4Job (Tan, Fong, and Bobroff, Tan et al.2010), SMP (Dai, Yang, and Zhang, Dai et al.2009), FTWeb (Santos, Lung, and Montez, Santos et al.2005). The three sub-figures show the different number of faults (2 faults, 5 faults and 8 faults) generated in SURETY. As can be seen, FLEX exhibits the lowest failure ratio in all cases due to the selection of the 'best' planning and replacement strategy.

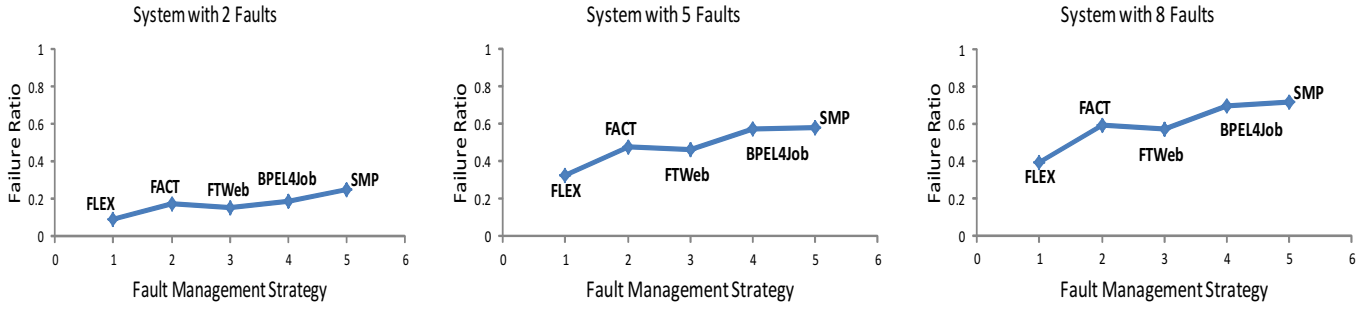


Figure 6.12: Impact of the number of faults on the failure ratio.

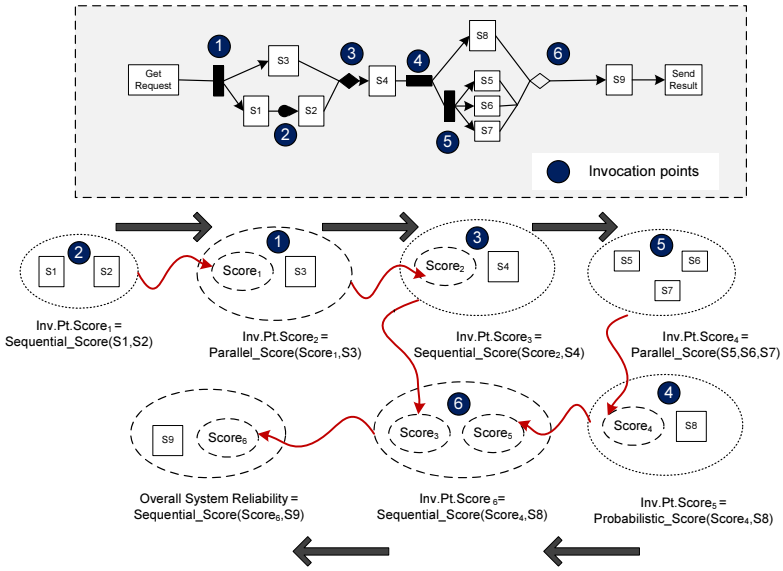


Figure 6.13: Assessing the over all system reliability.

As we mentioned earlier, the overall system reliability is assessed according to the reliability of each invocation point (Inv. Pt. in Figure 6.13). Our running example, includes nine Web services (S1,...,S9) and six invocation points (Inv.Pt$_1$,...,Inv.Pt$_6$). The running example also includes three similar Web services for each one of the nine Web services and we assign different scores to them. For example, the Web services (S11, S12, S13) are similar to S1, and each one of them has a different score (i.e., $Score_{S11} = 0.8$, $Score_{S12} = 0.5$, and $Score_{S13} = 0.7$). If S1 fails then we try to recover the system using one of the five recovery strategies. Figure 6.13 shows how the invocation point scores are calculated for our running scenario. The first step of our experiment is based on determining the invocation points and their respective sub invocation points. In the second step we calculate the score for each Web service involved in respective recovery planning strategy. The third step calculates the score for each Web service at every sub invocation point. The fourth step calculates the aggregated score for each invocation point and the final step calculates

the overall system reliability.

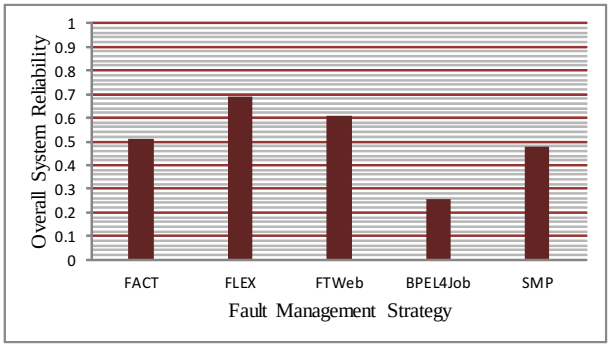


Figure 6.14: Overall reliability for different planning strategies.

We calculate the overall system reliability using Equation 31 for the above mentioned five different techniques in literature: FLEX, FACT (Liu, Li, Huang, and Xiao, Liu et al.2009), BPEL4Job (Tan, Fong, and Bobroff, Tan et al.2010), SMP (Dai, Yang, and Zhang, Dai et al.2009), FTWeb (Santos, Lung, and Montez, Santos et al.2005). Figure 6.14 shows the impacts of the five techniques on the overall reliability in our running scenario. As can be seen, FLEX has the highest overall reliability. This result is achieved by selecting the qualified (i.e., suitable) recovery plan based on the combination of QoS values of a service and users QoS requirements. FTWeb performs better incase the replicated services match user's preferences. While BPEL4Job has the worst overall reliability because it retries the same service (without much avail).

Figure 6.15 shows the probability of system faults for each planning strategy. BPEL4Job has the highest fault probability since we retry the same service and FLEX has the lowest probability of fault because the recovery plan selection is based on selecting the services with the lowest probability of failure. From the results shown in Figure 6.15, we can see that fault probability increases when any system ignores the user's preferences and the Web service performance (i.e.,

evaluating the selected Web service). However, the main factor that decreases the fault probability in FLEX is the fault prediction.

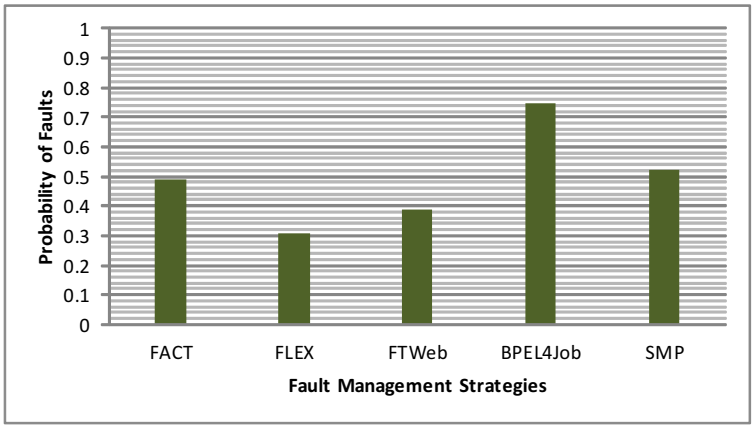


Figure 6.15: Probability of system faults for different planning strategies.

Figure 6.16a. shows a comparison between the Brute-force method (exhaustive search), Context-based matching (CBM) has been proposed in (Medjahed and Atif, Medjahed and Atif2007), Circular context-based (CCB) has been proposed in (Segev, Segev2008) and our similarity module (SM-FLEX) for service matching time based on the number of services. Note that Brute-force has the highest matching time especially when we have a large number of services. CME performs better than Brute-force method, but still takes more time than CCB. However, our method (SM-FLEX) provides the lowest service matching time, even if the number of services is large. Figure 6.16b. shows the relationship between the number of services and the search space for each matching method. We can see that Brute-force method has the largest search space and the smallest search is attributed to SM-FLEX.

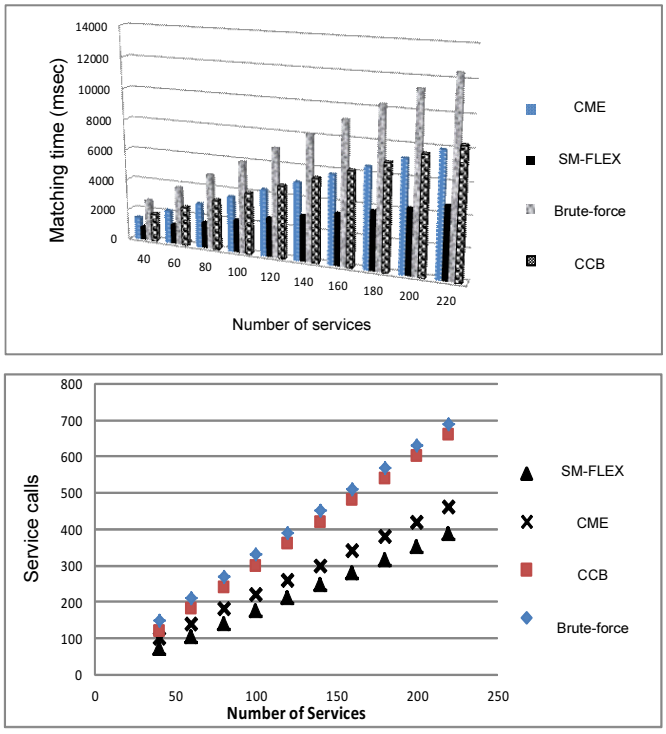


Figure 6.16: Matching Time and Search Space analysis.

6.4 Fault Propagation

In order to evaluate the performance of our proposed technique, we measured multiple factors: response time (Figure 6.18), failure ratio (Figure 6.19), total cost (Figure 6.20), fault propagation (Figure 6.21 and 6.22), overhead (Figure 6.23) and dropped messages (Figure 6.24). For this purpose, we used the the WSDream QoS-Dataset (Zheng and Lyu, Zheng and Lyu2010). This data-set contains 150 Web services distributed in computer nodes located all over the world (i.e., distributed in 22 different countries), where each Web service is invoked 100 times by a service user. Planet-Lab is employed for monitoring the Web services. The service users observe, collect, and contribute the failure data of the selected Web services. Our prototype is implemented in C# using Asp.Net running on Microsoft .Net version 3.5 and SQL as the back-end database.

Figure 6.17 shows the selected services for 40 compositions that implement our scenario. The compositions from 1 to 20 are alternatives for CS1 and the compositions from 21 to 40 are alternatives for CS2.

Alternative Compositions CS1	Flight Services	Hotel Services	Car Services	Movie Services	Museum Services	Alternative Compositions CS2
Composition 1	12.108.127.136	128.187.223.211	129.107.35.131	132.170.3.32	138.246.99.249	Composition 21
Composition 2	35.9.27.26	128.208.4.197	129.130.252.141	131.247.2.241	138.238.250.155	Composition 22
Composition 3	64.151.112.20	128.138.207.45	129.137.253.253	132.187.230.1	138.15.10.55	Composition 23
Composition 4	64.151.112.20	128.223.8.111	129.107.35.131	131.247.2.241	140.192.249.203	Composition 24
Composition 5	12.108.127.136	128.187.223.211	130.83.166.198	131.247.2.241	138.246.99.249	Composition 25
Composition 6	35.9.27.26	128.208.4.197	129.107.35.131	132.227.62.19	138.15.10.55	Composition 26
Composition 7	12.108.127.136	128.138.207.45	130.83.166.198	132.239.17.224	138.238.250.155	Composition 27
Composition 8	64.151.112.20	128.227.56.81	129.130.252.141	132.170.3.32	140.247.60.123	Composition 28
Composition 9	128.10.19.52	128.187.223.211	130.75.87.83	132.170.3.32	140.192.249.203	Composition 29
Composition 10	128.10.19.52	128.227.56.81	129.107.35.131	132.187.230.1	138.246.99.249	Composition 30
Composition 11	64.151.112.20	128.223.8.111	129.137.253.253	132.239.17.224	140.192.249.203	Composition 31
Composition 12	128.59.20.226	128.138.207.45	130.83.166.198	132.227.62.19	138.15.10.55	Composition 32
Composition 13	128.83.122.179	128.227.56.81	129.107.35.131	131.247.2.241	140.247.60.123	Composition 33
Composition 14	64.151.112.20	128.208.4.197	129.107.35.131	132.239.17.224	138.238.250.155	Composition 34
Composition 15	12.108.127.136	128.187.223.211	130.75.87.83	132.170.3.32	138.238.250.155	Composition 35
Composition 16	64.151.112.20	128.227.56.81	130.73.142.87	132.239.17.224	140.192.249.203	Composition 36
Composition 17	128.10.19.52	128.223.8.111	129.107.35.131	131.247.2.241	138.246.99.249	Composition 37
Composition 18	12.108.127.136	128.208.4.197	129.137.253.253	132.187.230.1	138.15.10.55	Composition 38
Composition 19	64.151.112.20	128.138.207.45	129.107.35.131	132.227.62.19	140.247.60.123	Composition 39
Composition 20	35.9.27.26	128.138.207.45	129.130.252.141	131.247.2.241	138.238.250.155	Composition 40

Figure 6.17: Service selection for multiple compositions

We compared our proposed approach on the overall response time of the composite solution. Figure 6.18 shows a response time comparison between the three different types of systems. First, the ideal system (i.e., system without any faults), second a system with faults but without using our soft-state protocol, and lastly system with faults that uses our soft-state protocol. We compared these three types of composite systems for the 40 compositions that are using the same car service. Using the system without faults as a base line, we can see that the system with faults and with soft-state has a better response time as compared to the system with faults and without soft-state.

We then observe the relationship of increasing user load and failure ratios among the composite

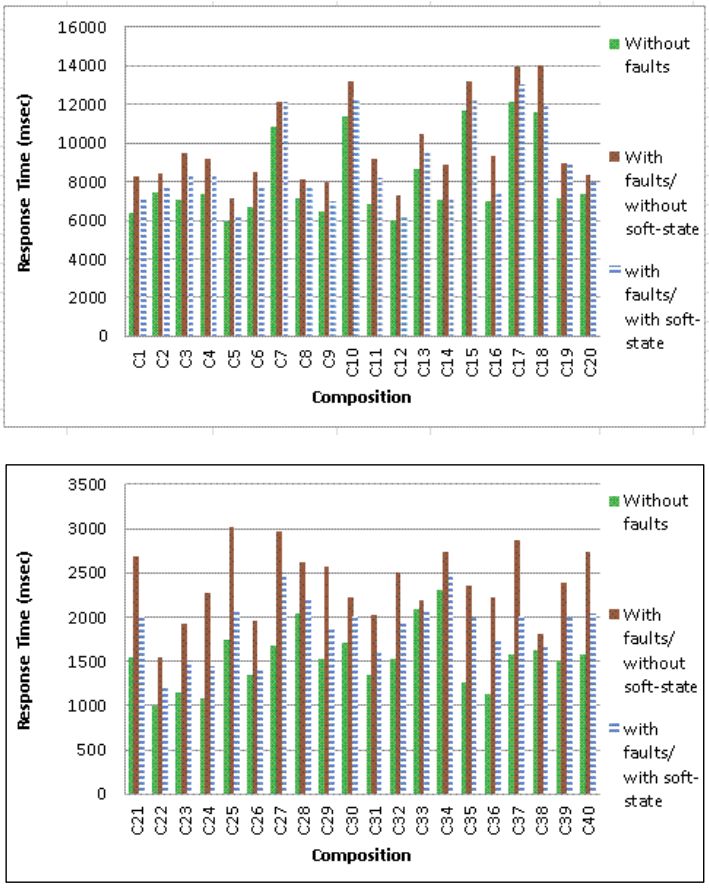


Figure 6.18: Service selection for multiple compositions

solutions, without using fault propagation (i.e., soft-state protocol). Figure6.19 shows the relationship between the failure ratio and execution time for different simultaneous user loads. When the number of users is increased, the failure ratio also increased. We can infer that is a multi-user scenario, if a fault occurs in any component service and the system does not support signaling protocols (i.e., protocols that that would inform other services/users about the failure), services cannot be notified of the faulty component and will come across this information once that faulty service is invoked. Hence The faulty service maybe used by another service/user, which will increase the probability of faults in the composite solution. To enrich our experiment, we create an analytical

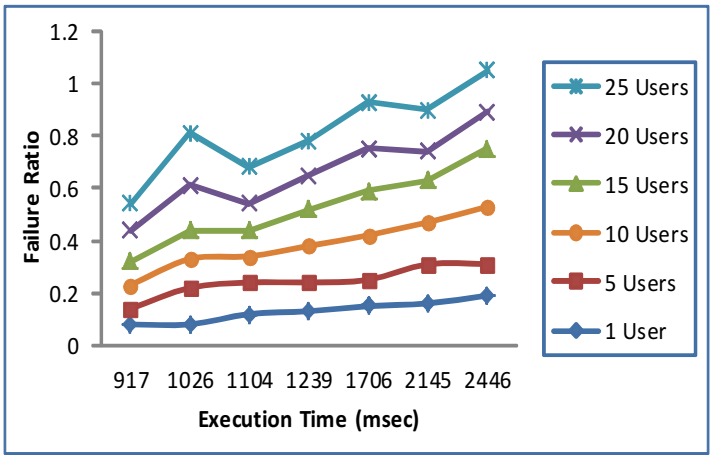


Figure 6.19: The relationship between failure ratio and execution time

module, table 6.7 defines the parameters and symbols used here after.

Table 6.7: Symbol Definition

Symbol	Definition
T	The total life time of a connection.
ET	Average life time of a connection.
λ	Message arrival rate.
TD	Teardown state.
Cr	Cost of each refresh message.
μ	Total number of refresh messages
TCr	Total cost of refresh messages.
P	The probability of a message getting lost.
Pr	The total loss probability for all messages.
Cs	The cost per unit time for being in an inconsistent state.
TCs	The total time of being in inconsistent states over the connection's entire life time.
$C(t_{SSR}, t_{SSS})$	The total cost.
N_{cs}	Number of context specifications per policy.
Cre	The re-initialization cost.

First, we analyze the relationship between t_{SSR}/t_{SSS} values and the following two parameters: fault propagation time and false faults. The total number of refresh messages that are sent during a connection's life time is the connection's life time divided by the refresh time which is the

summation of t_{SSR} and t_{SSS},

$$\mu \; = \; \frac{T}{(t_{SSR}, t_{SSS})} \tag{6.15}$$

The total cost of refresh messages during a connection's life time is the number of refresh messages times the cost of each refresh message,

$$TCr \; = \; Cr \times \mu \tag{6.16}$$

The total loss probability for all messages is loss probability for one message times the number of refresh messages,

$$Pr \; = \; P \times \mu \tag{6.17}$$

Since the cost per unit time of being in an inconsistent state is Cs then the total time of being in inconsistent states over the connection life time is

$$C(t_{SSR}, t_{SSS}) \; = \; TCr + TCs + Cre \times Pr \tag{6.18}$$

Then, the expected cost is then given as:

$$E[C(t_{SSR}, t_{SSS})] \; = \; E[TCr] + E[TCs] + E[Cre \times Pr] \tag{6.19}$$

Using the Equations 6.15, 6.16 and 6.17:

$$E[C(t_{SSR}, t_{SSS})] = E[Cr \times \frac{T}{(t_{SSR}, t_{SSS})}] + E[Cs \times P \times Pr] + E[Cre \times P \times \frac{T}{(t_{SSR}, t_{SSS})}] \quad (6.20)$$

Since the above mentioned values are based on the connection's life time (T), the expected value for the cost is given by:

$$E[C(t_{SSR}, t_{SSS})] = Cr \times \frac{E[T]}{(t_{SSR}, t_{SSS})} + Cs \times P \times E[T] + Cre \times P \times \frac{E[T]}{(t_{SSR}, t_{SSS})} \quad (6.21)$$

If the time for communication is T, the message arrival rate is λ, and there are n clients sending requests in different times within the interval [0,T], then the expected number of messages that reach the server are

$$N = 1 + n \times (\frac{T}{\lambda}) - (T \times \lambda)^n \quad (6.22)$$

Using the expected cost equation (Equation 6.21) and the expected number of messages at the server (Equation 6.22) we evaluate the total cost of our system with variable size window of refresh messages. Figure 6.20 presents the results of refresh window for 0.1 to 0.25 msec. In Figure 6.20(a)and (b) we can see that our technique is a still better than the hard state protocol for when the life time is small (10 and 20 msec). However, when we increase the life time to 40 msec there is a noticeable improvement of the cost for our technique. Figure 6.20(c) shows that our protocol perform better when we increase the life time of refresh message. Moreover,

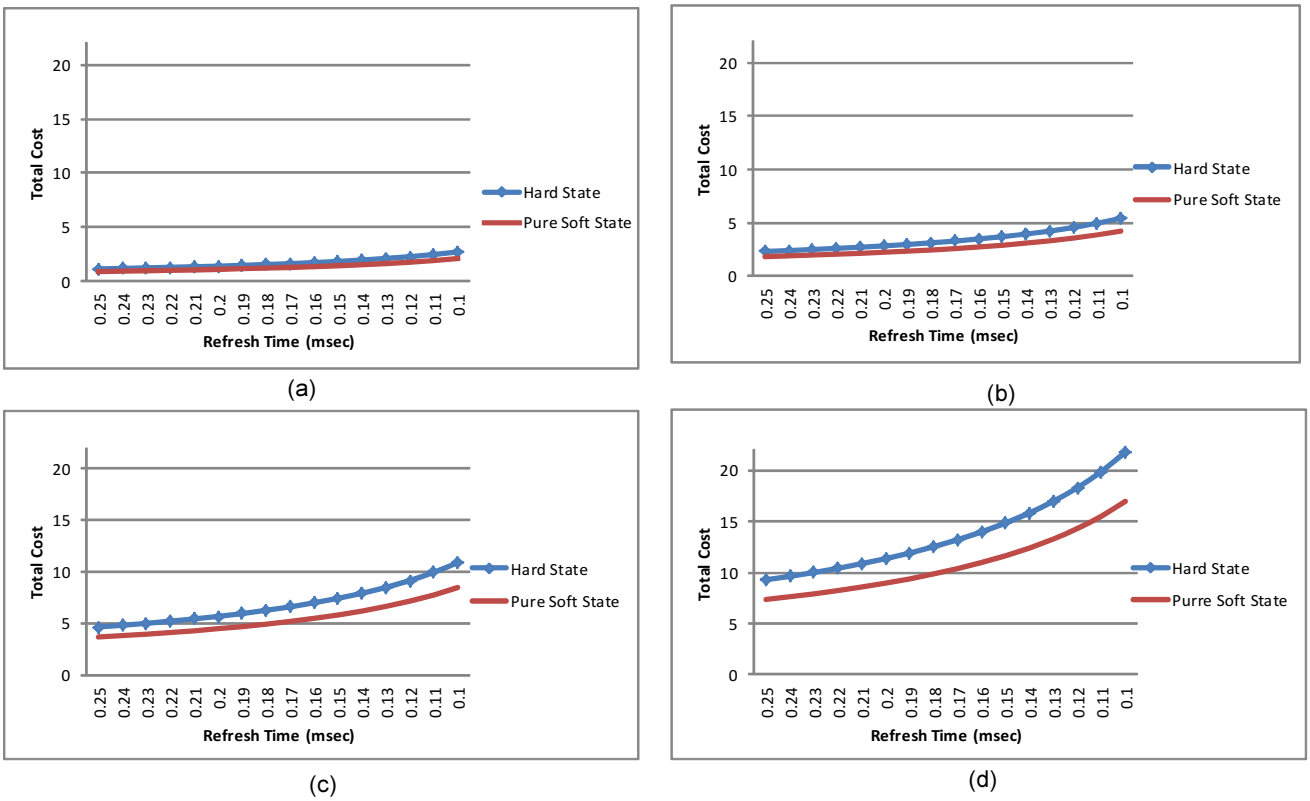


Figure 6.20: Total cost for life time (a) 10 msec (b) 20 msec (c) 40 msec (d) 80 msec

Figure 6.20(d) shows that when the life time is equal to 80, our method significantly reduces the overall cost of messages in the system. This shows that our method provides a better protocol in terms of message cost for all variable life time of refresh messages.

To analyze the Fault propagation time let us assume that a fault occurred in a sender service at time t_1 and has been detected by a receiver service at time t_2. The fault propagation time is equal to t_2-t_1 (i.e., the time it took for the receiver to detect a fault in its sender). Figure 6.21 compares the average fault propagation time for various t_{SSR} timer values. We consider different fault ratios for each t_{SSR} timer value. For instance, a fault ratio of 10 means that 10% (1 out of 10) of participants within a composite service failed. We focused on physical node faults; these are

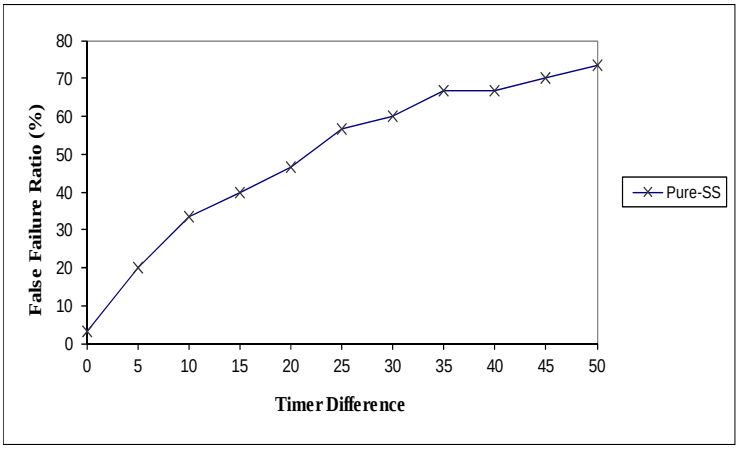


Figure 6.21: Impact of t_{SSR} on Fault Propagation Time.

created by physically stopping the services corresponding to faulty senders (selected randomly). Figure 6.21 shows that the t_{SSR} timer value has a direct impact on the fault propagation time, e.g., the smaller the t_{SSR}, the shorter is the fault propagation time. False faults refer to the situation where receiver service assume faults that did not actually occur in their corresponding sender services.

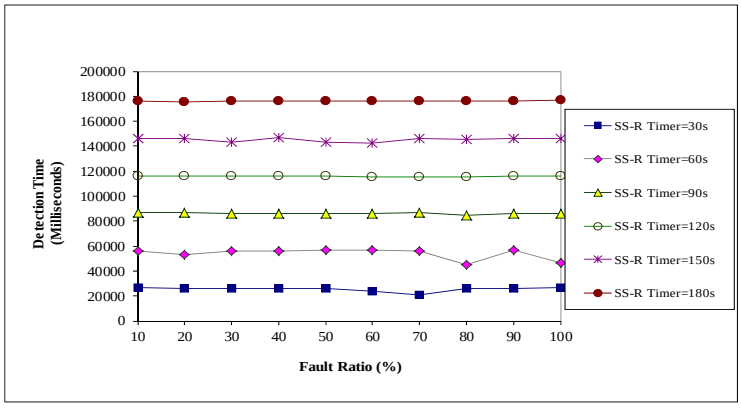
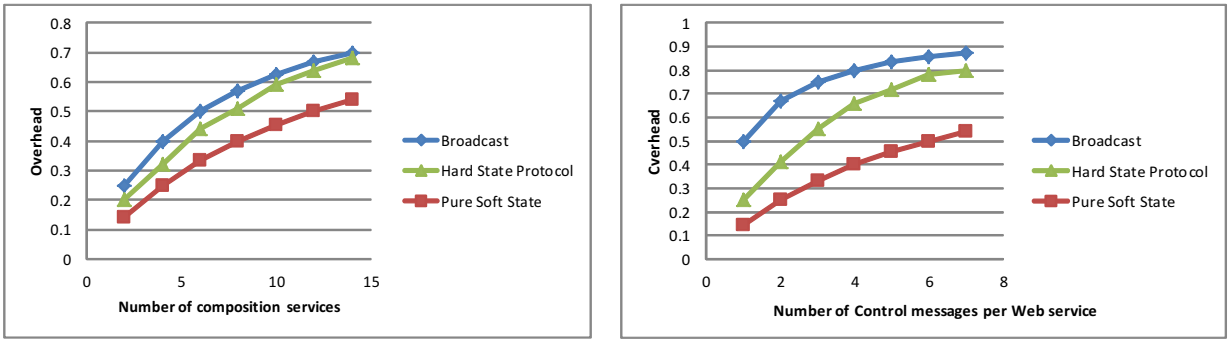


Figure 6.22: Relationship between t_{SSS} and t_{SSR} and impact on False Faults ratio.

Figure 6.22 depicts the relationship between false faults and time difference (i.e., $t_{SSS} - t_{SSR}$). We set t_{SSR} to 20s and vary t_{SSS} from 20s to 25s, 30s, 35s, 40s, etc. Figure 6.22. shows that false

faults occur if $t_{SSS} - t_{SSR} \geq 0$ (i.e., $t_{SSS} \geq t_{SSR}$). In addition, the bigger is t_{SSS} (compared to t_{SSR}), the larger is the number of false faults. These faults correspond to cases where Refresh() messages are sent after the end of the corresponding t_{SSR} cycles.



(a) (b)

Figure 6.23: Overhead in relation to the number of services and control messages.

We also evaluated our technique on the network traffic overhead. The overhead is calculated as number of control messages divided by the count of all messages generated by the system. The simulations show good improvement in the amount of overhead associated with our technique in comparison to the broadcast method and the hard state protocol. The broadcast means that the protocol require the server to receive acknowledgements from each client in the network even if it is not in the service/composition set, the server's processing capabilities become the bottleneck when the number of clients grows. Figure 6.23(a) shows that the overhead for broadcast technique and hard state are high and our technique decreases the overhead by more than 25%. In addition, we can see that our technique has a fairly low message overhead when we increase the number of composite services. Moreover, Figure 6.23(b) shows that the overhead for broadcast technique is much higher than our technique for the number of control messages per Web service.

The last set of the experiments depict cost of dropped messages by each protocol which can

be seen in Figure 6.24. Dropped message means that the node failed to send the required refresh message in a timely manner or the message did not reach its intended destination in due time. The information yielded in the figure shows that the pure soft-state is pretty consistent with the total cost even if the probability of dropped messages is high, while the hard state protocol has high cost as compared to our method.

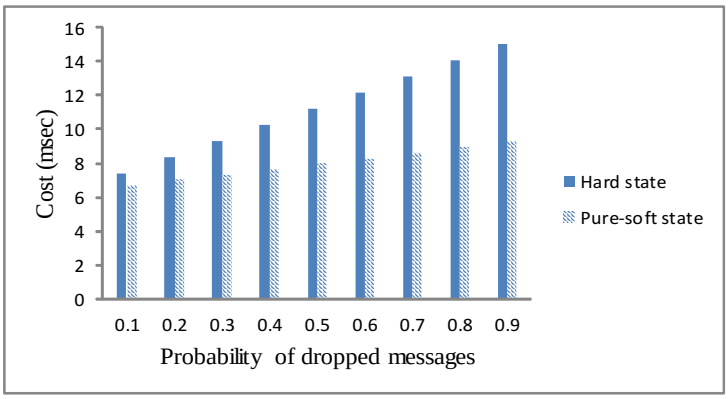


Figure 6.24: The extra system cost in relation to the probability of dropped messages.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, we predict the fault using the fault likelihood, assess the service reliability and based on this assessment we generate recovery plans on run time. Our main contributions include:

- We proposed a Fault Occurrence Likelihood Technique *FOLT* in SOAs.
- We extended FOLT to generate recovery plans in case of a fault (through the technique FauLt with EXception handling technique (FLEX)), using Quality of Service (QoS) definitions such as availability, cost, time, reputation, etc. to calculate the services' utility.
- We used the functional and non-functional properties of Web services to rank them for finding the 'best' Web services that are similar in case of replacement or replication (S^2R).
- We applied our similarity algorithm onto the Cloud environment with good results (Matching Alternative Services in the Cloud (SMART)).

Predicting Faults in Service-oriented Systems

The Web has started a steady evolution to become a vibrant environment (dubbed the Semantic Web) where applications can be automatically invoked by other Web clients. A key development in this regard has been the introduction of Web services. The ultimate goal of the Web services technology is enabling the use of independent components in a "composition" that is automatically formed as a result of the consumer's demand, and which may dissolve post demand-completion. Web services may make promises about the provided service and its associated quality but may

fail partially or fully to deliver these promises and bring down the quality of the whole application. Moreover, traditional fault management techniques may not totally support these Service-Oriented Architectures (SOAs) because of their autonomous and heterogeneous nature. My research addresses the fault management challenge by predicting the faults and calculating their occurrence likelihood in the system. I developed and implemented FOLT: a Fault Occurrence Likelihood estimation approach for service compositions. FOLT depends on three major factors: the service's past fault history, the time it takes to complete the required task in relation to the composition's total execution time, and the service's weight in the composition. Since a composed service using one or more of the invocation models (e.g., sequential, parallel, etc.), may encounter a fault during its execution, the likelihood of a fault's occurrence is directly proportional to the system's complexity. FOLT output values are thus influenced by the invocation model(s) used in the composition. In designing FOLT, we explored methods for the creation of a novel heuristics, its collection, assessment, and robustness of the system against malicious or incorrect information. The defined metrics are mathematically sound, and experiment results reveal that our approach provides a more robust, accurate, scalable, and relatively simple to deploy solution in comparison with existing works.

Semantic Similarity for SOA and Cloud

SOAs enable the automatic creation of business applications from independently developed and deployed services. Mechanisms are thus needed to select these service components that meet or exceed the functional and non-functional requirements of SOAs. The primary objective of service selection in SOAs can be viewed as a maximization of an application specific utility function that matches the constraints of the service requester against the capabilities and offerings of the

service provider(s). For this purpose, I developed and implemented S2R: a Semantic Web service Similarity and Ranking approach. Generally, similarity measurement consists of two components: syntactic and semantic. In S2R, both syntactic and semantic similarity filters are applied to find a set of Web services that match users' requirements. S2R is divided into three levels. In the first level, we filter the available Web services under a specific category based on their functional properties such as input, output and operations. In the second level, we further reduce the service search space based on non-functional properties, such as QoS parameters. Once a reduced pool of similar Web services is obtained, we rank them based on their utility value (in the third level). This value is calculated using a utility function which allows stakeholders to ascribe a value to the usefulness of the overall system as a function of several QoS attributes such as response time, availability, cost, reliability, etc. according to their preferences. Using the utility function, S2R filters Web services at each level so that more costly operations (e.g., reputation calculations) are applied on a reduced number of candidate services to shorten the time and space complexity of this search process. Moreover, since service selection is an on-demand process, we apply the S2R filters at run time. We compare S2R with similar existing approaches and the experiment results show the applicability and performance improvement in the service selection process, through time and search space complexity reduction.

Efficient Run-time Planning

Another aspect of my graduate research is fault-tolerance through self-healing. In this respect, I developed and implemented FLEX: FoLt with EXception handling technique. FLEX is a framework for infusing dependability in SOAs through self-healing. We identify a set of high-

level exception handling strategies based on the composition components' QoS performances and consumer requirements. Multiple recovery plans are produced and evaluated according to the performance of the included services, to select and execute the best recovery plan. We assess the overall system dependability using the generated plan and the available invocation information of the included services. FLEX combines our planning strategies based on FOLT calculations and incorporates the exception handling constructs of BPEL. Our planning module captures both the functional and non-functional features of Web services. Specifically, FLEX dynamically evaluates the performance of Web services based on their previous history (in terms of QoS), and user requirements. The likelihood of fault occurrence is then used to create (multiple) recovery plans. The best recovery plan is then chosen to be either executed immediately (if fault likelihood is above a pre-defined threshold), or saved for a later execution (i.e., to be executed when the fault occurs). The experiment results show that our proposed technique improves the service election quality by selecting the services with the highest score, and improves overall system performance.

There are still some limitations in predicting the service failures (i.e., the failure is the result of error/fault). Building efficient distributed systems that provide integration solutions (across multiple disciplines) while meeting expectations of reliability, performance and QoS requirements remains a challenging task. In my future research, I intend to leverage my graduate research experience in tackling these and related issues. First, I will continue and extend my research on FLEX; I plan to investigate reliability issues for SOAs and the Cloud. I plan to characterize FLEX behavior by integrating it to multiple real systems. As I start my career as a faculty member, my first major objective will be to write research proposals seeking funding from federal and other

agencies such as NSF. I plan to invest my best efforts towards securing the NSF CAREER grant to support my research. I plan to continue the current collaborations, and initiate new ones with faculty interested in my topics. Since fault management systems are relevant to various application domains, such as semantic Web and cloud computing, I see excellent collaboration opportunities with faculty members and industrial partners working in the above fields. Some items of immediate interests are:

- **Fault Management in the Cloud:** Redundancy is a fundamental concept in dependable computing which is applied to servers, software components, processors, storage, etc. There are two types of redundancy: time redundancy and space redundancy. With redundancy in time, operations are repeated several times whereas redundancy in space relies on redundant hardware and/or software. Both types increase the cost and lower the performance. Cloud computing has the potential to change this situation dramatically, since it first lowers the cost of redundant resources, second, offers to adjust resources dynamically, and third, allows temporarily the use a huge amount of resources. However, the Cloud still suffers from many challenges in using the current fault management techniques. Thus, I intend to study the main factors of increasing the performance of the Cloud, implement my defined techniques on the Cloud, starting from predicting faults, determining the best recovery strategy for each situation and improving the system performance (i.e., reduce cost, reduce time, guarantee availability, and increase reliability).

- **Decentralized Automatic Management:** Reflection is the ability of a system to monitor and change its own behavior, as well as aspects of its implementation and allowing the ability to

be sensitive to its environment. To achieve high level of reflection, I will build an automatic and decentralized fault management approach. The new approach will be in generic enough to be implemented in a heterogeneous environment.

- **Trustworthy Service Compositions:** The sharing of privately owned data through and across remote applications while respecting access control restrictions poses a fundamental challenge in building loosely coupled distributed systems (e.g. Web services compositions). I intend to explore how services hosted at different sites can be composed while respecting data access requirements. I plan to investigate the applicability of cryptographic techniques (such as private information retrieval) and reputation in enforcing these requirements.

- **Accurate Reputation:** A fundamental issue in evaluating any Web service is determining other Web services reputation accuracy. Our work is primarily dependent on the reputation that is provided from other services. For example, a technical incident at a server running a Web service may cause the service to be unavailable. The unavailability may prompt the service requester to change the reputation rating immediately, decide to wait for a period of time before updating the reputation rating, or put the suspected service on probation as a temporary measure. I intend to explore techniques for accurate reputation change management so that no service provider is wrongfully disadvantaged.

BIBLIOGRAPHY

ABRAMOWICZ, W., HANIEWICZ, K., KACZMAREK, M., AND ZYSKOWSKI, D. 2007. Architecture for web services filtering and clustering. *Internet and Web Applications and Services, International Conference on 0*, 18.

ACKOFF, R. L. 1978. *Redesigning the future*. Wiley, New York.

AGHDAIE, N. AND TAMIR, Y. 2009. Coral: A transparent fault-tolerant web service. *J. Syst. Softw. 82,* 1, 131–143.

ALHOSBAN, A., HASHMI, K., MALIK, Z., AND MEDJAHED, B. 2011. Assessing fault occurrence likelihood for service-oriented systems. In *Proceedings of the 11th International Conference on Web Engineering*. 59–73.

ALHOSBAN, A., HASHMI, K., MALIK, Z., AND MEDJAHED, B. 2012. S2r: A semantic web service similarity and ranking approach. *INTERNATIONAL JOURNAL OF NEXT-GENERATION COMPUTING 3,* 2.

ALONSO, G., CASATI, F., KUNO, H. A., AND MACHIRAJU, V. 2004. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer.

ALRIFAI, M., SKOUTAS, D., AND RISSE, T. 2010. Selecting skyline services for qos-based web service composition. In *Proceedings of the 19th international conference*. WWW '10. ACM, New York, USA, 11–20.

ANDREWS, T., CURBERA, F., DHOLAKIA, H., GOLAND, Y., KLEIN, J., LEYMANN, F., LIU, K., ROLLER, D., SMITH, D., THATTE, S., TRICKOVIC, I., AND WEERAWARANA, S. 2003. *BPEL4WS, Business Process Execution Language for Web Services Version 1.1*. IBM.

ANDRIKOPOULOS, V., BENBERNOU, S., AND PAPAZOGLOU, M. P. 2008. Managing the evolution of service specifications. In *Proceedings of the 20th international conference on Advanced Information Systems Engineering*. CAiSE '08. Springer-Verlag, Berlin, Heidelberg, 359–374.

ARDISSONO, L., CONSOLE, L., GOY, A., PETRONE, G., PICARDI, C., SEGNAN, M., AND DUPRÉ, D. T. 2005. Advanced fault analysis in web service composition. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*. ACM, New York, NY, USA, 1090–1091.

BAI, C. G., HU, Q. P., XIE, M., AND NG, S. H. 2005. Software failure prediction based on a markov bayesian network model. *J. Syst. Softw. 74,* 3, 275–282.

BAILEY, J., POULOVASSILIS, A., AND WOOD, P. T. 2002. An event-condition-action language for xml. In *Proceedings of the 11th international conference on World Wide Web*. WWW '02. ACM, New York, NY, USA, 486–495.

BAÏNA, K., BENALI, K., AND GODART, C. 2001. A process service model for dynamic enterprise process interconnection. In *Proceedings of the 9th International Conference on Cooperative Information Systems*. CooplS '01. Springer-Verlag, London, UK, 239–254.

BEN LAKHAL, N., KOBAYASHI, T., AND YOKOTA, H. 2009. Fenecia: failure endurable nested-transaction based execution of composite web services with incorporated state analysis. *The VLDB Journal 18,* 1, 1–56.

BERGMANN, R., RICHTER, M. M., SCHMITT, S., STAHL, A., AND VOLLRATH, I. 2001. Utility-oriented matching: A new research direction for case-based reasoning. In *In professionlles wissens managment: Erfahrungen Und Visionen. Proceeding of the 1st conference knowledgr management. Shaker*. 264–274.

BOOTH, D. AND LIU, C. K. 2006. Web services description language (wsdl). http://w3.org/TR/2006/CR-wsdl20-primer-20060327/.

BOUGUETTAYA, A., KRÜGER, I., AND MARGARIA, T., Eds. 2008. *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings*. Lecture Notes in Computer Science Series, vol. 5364.

BRAMBILLA, M., CERI, S., COMAI, S., AND TZIVISKOU, C. 2005. Exception handling in workflow-driven web applications. In *Proceedings of the 14th international conference on World Wide Web*. WWW '05. ACM, New York, NY, USA, 170–179.

BURNS, A. AND WELLINGS, A. J. 2001. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BUYYA, R., YEO, C. S., VENUGOPAL, S., BROBERG, J., AND BRANDIC, I. 2009. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst. 25*, 599–616.

CARDOSO, J., MILLER, J., SHETH, A., AND ARNOLD, J. 2002. Modeling quality of service for workflows and web service processes. *Journal of Web Semantics 1*, 281–308.

CASTRO, M. AND LISKOV, B. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20,* 4, 398–461.

CASTRO, M., RODRIGUES, R., AND LISKOV, B. 2003. Base: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst. 21,* 3, 236–269.

CHAFLE, G., DASGUPTA, K., KUMAR, A., MITTAL, S., AND SRIVASTAVA, B. 2006. Adaptation inweb service composition and execution. *Web Services, IEEE International Conference on 0*, 549–557.

CHAKRABORTY, D., PERICH, F., JOSHI, A., FININ, T. W., AND YESHA, Y. 2002. A reactive service composition architecture for pervasive computing environments. In *Proceedings of the IFIP TC6/WG6.8 Working Conference on Personal Wireless Communications*. PWC '02. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 53–62.

CHAN, K. S. M. AND BISHOP, J. 2009. The design of a self-healing composition cycle for web services. In *SEAMS '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Computer Society, Washington, DC, USA, 20–27.

CHAN, N. N., GAALOUL, W., AND TATA, S. 2011. A web service recommender system using vector space model and latent semantic indexing. *Advanced Information Networking and Applications, International Conference on 0*, 602–609.

CHANDRA, S., ELLIS, C. S., AND VAHDAT, A. 2000. Differentiated multimedia web services using quality aware transcoding. In *INFOCOM*. 961–969.

CHENG, S.-T., LI, S.-Y., AND CHEN, C.-M. 2008. Distributed detection in wireless sensor networks. *Computer and Information Science, ACIS International Conference on 0*, 401–406.

COMUZZI, M. AND PERNICI, B. 2009. A framework for qos-based web service contracting. *ACM Trans. Web 3*, 10:1–10:52.

DAI, Y., YANG, L., AND ZHANG, B. 2009. Qos-driven self-healing web service composition based on performance prediction. *J. Comput. Sci. Technol. 24*, 250–261.

DEMSKY, B. AND RINARD, M. 2003. Automatic data structure repair for self-healing systems. In *In Proceedings of the 1 st Workshop on Algorithms and Architectures for Self-Managing Systems*.

DENARO, G., PEZZÉ, M., TOSI, D., AND SCHILLING, D. 2006. Towards self-adaptive service-oriented architectures. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*. ACM, New York, NY, USA, 10–16.

DEY, A. K. 2000. Providing architectural support for building context-aware applications. Ph.D. thesis, Atlanta, GA, USA. AAI9994400.

DIALANI, V., MILES, S., MOREAU, L., ROURE, D. D., AND LUCK, M. 2002. Transparent fault tolerance for web services based architectures. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*. Springer-Verlag, London, UK, 889–898.

D'MELLO, D. A. AND ANANTHANARAYANA, V. S. 2009. A tree structure for web service compositions. In *COMPUTE '09: Proceedings of the 2nd Bangalore Annual Compute Conference*. ACM, New York, NY, USA, 1–4.

DUDLEY, G., JOSHI, N., OGLE, D. M., SUBRAMANIAN, B., AND TOPOL, B. B. 2004. Autonomic self-healing systems in a cross-product it environment. *Autonomic Computing, International Conference on 0*, 312–313.

ERRADI, A., MAHESHWARI, P., AND TOSIC, V. 2007. Ws-policy based monitoring of composite web services. In *Proceedings of the Fifth European Conference on Web Services*. IEEE Computer Society, Washington, DC, USA, 99–108.

FANG, C.-L., LIANG, D., LIN, F., AND LIN, C.-C. 2007. Fault tolerant web services. *J. Syst. Archit. 53*, 21–38.

FARATIN, P., SIERRA, C., AND JENNINGS, N. R. 2002. Using similarity criteria to make issue trade-offs in automated negotiations. *Artif. Intell. 142,* 2, 205–237.

FOGG, B. J. AND ECKLES, D. 2007. The behavior chain for online participation: how successful web services structure persuasion. In *Proceedings of the 2nd international conference on Persuasive technology*. PERSUASIVE'07. Springer-Verlag, Berlin, Heidelberg, 199–209.

FRANK, D., LAM, L., FONG, L., FANG, R., AND KHANGAONKAR, M. 2008. Using an interface proxy to host versioned web services. In *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*. SCC '08. IEEE Computer Society, Washington, DC, USA, 325–332.

GADGIL, H., FOX, G., PALLICKARA, S., AND PIERCE, M. 2007. Scalable, fault-tolerant management in a service oriented architecture. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*. ACM, New York, NY, USA, 235–236.

GARLAN, D. AND SCHMERL, B. 2002. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*. WOSS '02. ACM, New York, NY, USA, 27–32.

GHOSH, D., SHARMAN, R., RAGHAV RAO, H., AND UPADHYAYA, S. 2007. Self-healing systems - survey and synthesis. *Decis. Support Syst. 42,* 4, 2164–2185.

GRIFFITH, R., KAISER, G., AND LÓPEZ, J. A. 2009. Multi-perspective evaluation of self-healing systems using simple probabilistic models. In *ICAC '09: Proceedings of the 6th international conference on Autonomic computing*. ACM, New York, NY, USA, 59–60.

GU, X., NAHRSTEDT, K., YUAN, W., WICHADAKUL, D., AND XU, D. 2001. An xml-based quality of service enabling language for the web. Tech. rep., Champaign, IL, USA.

GU, Z., LI, J., TANG, J., XU, B., AND HUANG, R. 2007. Verification of web service conversations specified in wscl. In *Proceedings of the 31st Annual International Computer Software*

and Applications Conference - Volume 02. COMPSAC '07. IEEE Computer Society, Washington, DC, USA, 432–437.

GUINEA, S. 2005. Self-healing web service compositions. In *Proceedings of the 27th international conference on Software engineering*. ICSE '05. ACM, New York, NY, USA, 655–655.

GUOPING, Z., HUIJUAN, Z., AND ZHIBIN, W. 2009. A qos-based web services selection method for dynamic web service composition. In *Proceedings of the 2009 First International Workshop on Education Technology and Computer Science - Volume 03*. ETCS '09. IEEE Computer Society, Washington, DC, USA, 832–835.

HALIMA, R. B., DRIRA, K., AND JMAIEL, M. 2008. A qos-oriented reconfigurable middleware for self-healing web services. In *Proceedings of the 2008 IEEE International Conference on Web Services*. ICWS '08. IEEE Computer Society, Washington, DC, USA, 104–111.

HANEMANN, A. 2006. A hybrid rule-based/case-based reasoning approach for service fault diagnosis. In *AINA (2)*. 734–740.

HASHMI, K., ALHOSBAN, A., MALIK, Z., AND MEDJAHED, B. 2011. Webneg: A genetic algorithm based approach for service negotiation. In *ICWS*. 105–112.

HEUVEL, W.-J. V. D., YANG, J., AND PAPAZOGLOU, M. P. 2001. Service representation, discovery, and composition for e-marketplaces. In *Proceedings of the 9th International Conference on Cooperative Information Systems*. CooplS '01. Springer-Verlag, London, UK, 270–284.

HUANG, X., ZOU, S., WANG, W., AND CHENG, S. 2005. Mdfm: Multi-domain fault management for internet services. In *MMNS'05*. 121–132.

HWANG, S.-Y., WANG, H., TANG, J., AND SRIVASTAVA, J. 2007. A probabilistic approach to modeling and estimating the qos of web-services-based workflows. *Inf. Sci. 177,* 23, 5484–5503.

IBRAHIM, D. H. A. 2009. The concept of web service versioning in provenance. In *Proceedings of the 2009 International Conference on Network-Based Information Systems*. NBIS '09. IEEE Computer Society, Washington, DC, USA, 469–474.

JACQUES-SILVA, G., CHALLENGER, J., DEGENARO, L., GILES, J., AND WAGLE, R. 2008. Self healing in system-s. *Cluster Computing 11,* 3, 247–257.

JENSEN, R. M. 2004. Fault tolerant planning: Toward probabilistic uncertainty models in symbolic non-deterministic planning. In *In Proceedings of the 14th International Conference on Automated Planning and Scheduling ICAPS-04*. 335–344.

JI, P., GE, Z., KUROSE, J., AND TOWSLEY, D. 2007. A comparison of hard-state and soft-state signaling protocols. *IEEE/ACM Trans. Netw. 15,* 2, 281–294.

KARIMZADEHGAN, M., LI, W., ZHANG, R., AND MAO, J. 2011. A stochastic learning-to-rank algorithm and its application to contextual advertising. In *Proceedings of the 20th international conference on World wide web*. WWW '11. ACM, New York, NY, USA, 377–386.

KATCHABAW, M. J., LUTFIYYA, H. L., MARSHALL, A. D., AND BAUER, M. A. 1996. Policy-driven fault management in distributed systems. In *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering*. IEEE Computer Society, Washington, DC, USA, 236.

KELLNER, I. AND FIEGE, L. 2009. Viewpoints in complex event processing: industrial experience report. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS '09. ACM, New York, NY, USA, 9:1–9:8.

KEROMYTIS, A. D. 2007. Characterizing self-healing software systems. In *Proceedings of the 4th International Conference on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS*.

KOKASH, N. 2007. Risk management for service-oriented systems. In *ICWE'07: Proceedings of the 7th international conference on Web engineering*. Springer-Verlag, Berlin, Heidelberg, 563–568.

KONTOGIANNIS, K., LEWIS, G. A., SMITH, D. B., LITOIU, M., MULLER, H., SCHUSTER, S., AND STROULIA, E. 2007. The landscape of service-oriented systems: A research perspective. *Systems Development in SOA Environments, International Workshop on 0*, 1.

KOTLA, R., CLEMENT, A., WONG, E., ALVISI, L., AND DAHLIN, M. 2008. Zyzzyva: speculative byzantine fault tolerance. *Commun. ACM 51,* 11, 86–95.

KOUADRI MOSTÉFAOUI, G. AND BRÉZILLON, P. 2006. Context-based constraints in security: Motivations and first approach. *Electron. Notes Theor. Comput. Sci. 146*, 85–100.

KRISHNAMURTHY, V. AND BABU, C. 2012. Pattern based adaptation for service oriented applications. *SIGSOFT Softw. Eng. Notes 37,* 1, 1–6.

LAKHAL, N. B., KOBAYASHI, T., AND YOKOTA, H. 2006. Dependability and flexibility centered approach for composite web services modeling. In *OTM Conferences (1)*. 163–182.

LALA, P. K. AND KUMAR, B. K. 2002. An architecture for self-healing digital systems. In *IOLTW '02: Proceedings of the Proceedings of The Eighth IEEE International On-Line Testing Workshop (IOLTW'02)*. IEEE Computer Society, Washington, DC, USA, 3.

LASTER, S. S. AND OLATUNJI, A. O. 2007. Abstract autonomic computing: Towards a self-healing system.

LEE, C. AND HELAL, S. 2003. Context attributes: An approach to enable context-awareness for service discovery. In *Proceedings of the 2003 Symposium on Applications and the Internet*. SAINT '03. IEEE Computer Society, Washington, DC, USA, 22–.

LEE, Y. 2011. bqos(business qos) parameters for soa quality rating. In *FGIT-ASEA/DRBC/EL*. 497–504.

LI, B., XU, Y., WU, J., AND ZHU, J. 2012. A petri-net and qos based model for automatic web service composition. *JSW 7,* 1, 149–155.

LI, L. AND HORROCKS, I. 2003. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th international conference*. WWW '03. ACM, New York, USA, 331–339.

LIANG, D., LO, W.-T., KAO, Y., YUAN, S., AND CHANG, Y.-S. 1997. A fault tolerant object transaction service in corba. *Computer Software and Applications Conference, Annual International 0*, 115.

LIN, C., LU, S., LAI, Z., CHEBOTKO, A., FEI, X., HUA, J., AND FOTOUHI, F. 2008. Service-oriented architecture for view: A visual scientific workflow management system. In *SCC '08:*

Proceedings of the 2008 IEEE International Conference on Services Computing. IEEE Computer Society, Washington, DC, USA, 335–342.

LIU, A., LI, Q., HUANG, L., AND XIAO, M. 2009. Facts: A framework for fault-tolerant composition of transactional web services. *IEEE Transactions on Services Computing 99,* PrePrints, 46–59.

LIU, A., LI, Q., HUANG, L., AND XIAO, M. 2010. Facts: A framework for fault-tolerant composition of transactional web services. *IEEE Transactions on Services Computing 99,* PrePrints, 46–59.

LUTFIYYA, H. L., BAUER, M. A., MARSHALL, A. D., AND STOKES, D. K. 2000. Fault management in distributed systems: A policy-driven approach. *J. Netw. Syst. Manage. 8,* 4, 499–525.

LYU, M. R. 2007. Software reliability engineering: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*. IEEE Computer Society, Washington, DC, USA, 153–170.

MAAMAR, Z., BENSLIMANE, D., AND NARENDRA, N. C. 2006. What can context do for web services? *Commun. ACM 49*, 98–103.

MAJZIK, I. AND HUSZERL, G. 2002. Towards dependability modeling of ft-corba architectures. In *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*. EDCC-4. Springer-Verlag, London, UK, 121–139.

MALIK, Z., AKBAR, I., AND BOUGUETTAYA, A. 2009. Web services reputation assessment using a hidden markov model. In *Proceedings of the 7th International Joint Conference on Service-Oriented Computing*. ICSOC-ServiceWave '09. Springer-Verlag, Berlin, Heidelberg, 576–591.

MALIK, Z. AND BOUGUETTAYA, A. 2009. Rateweb: Reputation assessment for trust establishment among web services. *The VLDB Journal 18,* 4, 885–911.

MARTIN, D., BURSTEIN, M., MCDERMOTT, D., MCILRAITH, S., PAOLUCCI, M., SYCARA, K., MCGUINNESS, D. L., SIRIN, E., AND SRINIVASAN, N. 2007. Bringing semantics to web services with owl-s. *World Wide Web 10*, 243–277.

MAUREL, Y., DIACONESCU, A., AND LALANDA, P. 2010. Ceylon: A service-oriented framework for building autonomic managers. *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on 0*, 3–11.

MECELLA, M., PERNICI, B., AND CRACA, P. 2001. Compatibility of e -services in a cooperative multi-platform environment. In *Proceedings of the Second International Workshop on Technologies for E-Services*. TES '01. Springer-Verlag, London, UK, 44–57.

MEDJAHED, B. AND ATIF, Y. 2007. Context-based matching for web service composition. *Distrib. Parallel Databases 21*, 5–37.

MEDJAHED, B. AND MALIK, Z. 2011. Bottom-up fault management in composite web services. In *CAiSE*. 597–611.

MENASCE, D. A. 2004. Composing web services: A qos view. *IEEE Internet Computing 8*, 88–90.

MENASCÉ, D. A. AND DUBEY, V. K. 2007. Utility-based qos brokering in service oriented architectures. In *ICWS*. 422–430.

MERIDETH, M. G., IYENGAR, A., MIKALSEN, T., ROUVELLOU, I., AND NARASIMHAN, P. 2005. Thema: Byzantine-fault-tolerant middleware for web services applications. In *In Proceedings of the IEEE Symposium on Reliable Distributed Systems*. IEEE Computer Society, 131–142.

MEULENHOFF, P. J., OSTENDORF, D. R., ŽIVKOVIĆ, M., MEEUWISSEN, H. B., AND GIJSEN, B. M. 2009. Intelligent overload control for composite web services. In *ICSOC-ServiceWave '09: Proceedings of the 7th International Joint Conference on Service-Oriented Computing*. Springer-Verlag, Berlin, Heidelberg, 34–49.

MODAFFERI, S., MUSSI, E., AND PERNICI, B. 2006. Sh-bpel: a self-healing plug-in for ws-bpel engines. In *MW4SOC '06: Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*. ACM, New York, NY, USA, 48–53.

MORGAN, G., SHRIVASTAVA, S. K., EZHILCHELVAN, P. D., AND LITTLE, M. C. 1999. Design and implemantation of a corba fault-tolerant object group service. In *Proceedings of the IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems II*. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 361–374.

MOZAFARI, B., ZENG, K., AND ZANIOLO, C. 2012. High-performance complex event processing over xml streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. ACM, New York, NY, USA, 253–264.

MULO, E., ZDUN, U., AND DUSTDAR, S. 2010. An event view model and dsl for engineering an event-based soa monitoring infrastructure. In *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, New York, NY, USA, 62–72.

NARAYANAN, S. AND MCILRAITH, S. A. 2002. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference*. WWW '02. ACM, New York, USA, 77–88.

NASCIMENTO, A. S., RUBIRA, C. M. F., AND LEE, J. 2011. An spl approach for adaptive fault tolerance in soa. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*. SPLC '11. ACM, New York, NY, USA, 15:1–15:8.

NEJATI, S., SABETZADEH, M., CHECHIK, M., EASTERBROOK, S., AND ZAVE, P. 2007. Matching and merging of statecharts specifications. In *Proceedings of the 29th international conference on Software Engineering*. ICSE '07. IEEE Computer Society, Washington, DC, USA, 54–64.

NEPAL, S., SHERCHAN, W., HUNKLINGER, J., AND BOUGUETTAYA, A. 2010. A fuzzy trust management framework for service web. In *ICWS*. 321–328.

PAOLUCCI, M. AND WAGNER, M. 2006. Grounding owl-s in wsdl-s. In *Proceedings of the IEEE International Conference on Web Services*. IEEE Computer Society, Washington, DC, USA, 913–914.

PAPAMARKOS, G., POULOVASSILIS, A., AND WOOD, P. T. 2011. Performance modelling of event-condition-action rules in p2p networks. *J. Comput. Syst. Sci. 77,* 4, 621–636.

PAPAZOGLOU, M. P., POHL, K., PARKIN, M., AND METZGER, A., Eds. 2010. *Service Research Challenges and Solutions for the Future Internet - S-Cube - Towards Engineering, Managing and Adapting Service-Based Systems*. Lecture Notes in Computer Science Series, vol. 6500. Springer.

PARADIS, L. AND HAN, Q. 2007. A survey of fault management in wireless sensor networks. *J. Netw. Syst. Manage. 15,* 2, 171–190.

PARK, J., YOUN, H., AND LEE, E. 2009. An automatic code generation for self-healing. *J. Inf. Sci. Eng. 25,* 6, 1753–1781.

PASCALAU, E. AND GIURCA, A. 2009. Towards enabling saas for business rules. In *BPSC*. 207–222.

PERNICI, B. AND SIADAT, S. H. 2011. Adaptation of web services based on qos satisfaction. In *Proceedings of the 2010 international conference*. ICSOC'10. Springer-Verlag, Berlin, Heidelberg, 65–75.

POULOVASSILIS, A., PAPAMARKOS, G., AND WOOD, P. T. 2006. Event-condition-action rule languages for the semantic web. In *Proceedings of the 2006 international conference on Current Trends in Database Technology*. EDBT'06. Springer-Verlag, Berlin, Heidelberg, 855–864.

QIAO, Y., ZHONG, K., WANG, H., AND LI, X. 2007. Developing event-condition-action rules in real-time active database. In *Proceedings of the 2007 ACM symposium on Applied computing*. SAC '07. ACM, New York, NY, USA, 511–516.

RAZ, O., KOOPMAN, P., AND SHAW, M. 2002. Semantic anomaly detection in online data sources. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. ACM, New York, NY, USA, 302–312.

RESINAS, M., FERNANDEZ, P., AND CORCHUELO, R. 2012. A bargaining-specific architecture for supporting automated service agreement negotiation systems. *Sci. Comput. Program. 77,* 1, 4–28.

ROBINSON, D. AND KOTONYA, G. 2008. A self-managing brokerage model for quality assurance in service-oriented systems. *High-Assurance Systems Engineering, IEEE International Symposium on 0*, 424–433.

ROMAN, D. AND KIFER, M. 2007. Reasoning about the behavior of semantic web services with concurrent transaction logic. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB '07. VLDB Endowment, 627–638.

SALAS, J., PEREZ-SORROSAL, F., PATI NO-MARTÍNEZ, M., AND JIMÉNEZ-PERIS, R. 2006. Ws-replication: a framework for highly available web services. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*. ACM, New York, NY, USA, 357–366.

SALFNER, F., SCHIESCHKE, M., AND MALEK, M. 2006. Predicting failures of computer systems: a case study for a telecommunication system. In *Proceedings of the 20th international conference on Parallel and distributed processing*. IPDPS'06. IEEE Computer Society, Washington, DC, USA, 348–348.

SANTOS, G. T., LUNG, L. C., AND MONTEZ, C. 2005. Ftweb: A fault tolerant infrastructure for web services. In *Proceedings of the IEEE International Enterprise Computing Conference*. IEEE Computer Society, 95–105.

SEGEV, A. 2008. Circular context-based semantic matching to identify web service composition. In *Proceedings of the 2008 international workshop on Context enabled source and service selection, integration and adaptation: organized with the 17th International World Wide Web Conference (WWW 2008)*. CSSSIA '08. ACM, New York, NY, USA, 7:1–7:5.

SHERCHAN, W., NEPAL, S., HUNKLINGER, J., AND BOUGUETTAYA, A. 2010. A trust ontology for semantic services. In *IEEE SCC*. 313–320.

SIMMONDS, J., GAN, Y., CHECHIK, M., NEJATI, S., O'FARRELL, B., LITANI, E., AND WATERHOUSE, J. 2009. Runtime monitoring of web service conversations. *IEEE Transactions on Services Computing 99,* PrePrints, 223–244.

SINGH, A., FONSECA, P., KUZNETSOV, P., RODRIGUES, R., AND MANIATIS, P. 2009. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. USENIX Association, Berkeley, CA, USA, 169–184.

SOAP. 2007. Soap version 1.2. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

SYCARA, K., KLUSCH, M., WIDOFF, S., AND LU, J. 1999. Dynamic service matchmaking among agents in open information environments. *SIGMOD Rec. 28*, 47–53.

TAN, W., FONG, L., AND BOBROFF, N. 2010. Bpel4job: A fault-handling design for job flow management. In *Proceedings of the 5th international conference on Service-Oriented Computing*. ICSOC '07. Springer-Verlag, Berlin, Heidelberg, 27–42.

VACULIN, R., WIESNER, K., AND SYCARA, K. 2008. Exception handling and recovery of semantic web services. *Networking and Services, International conference on 0*, 217–222.

VERMA, K. AND SHETH, A. P. 2005. Autonomic web processes. In *Proceedings of the Third international conference on Service-Oriented Computing*. ICSOC'05. Springer-Verlag, Berlin, Heidelberg, 1–11.

WANG, M., BANDARA, K. Y., AND PAHL, C. 2009. Integrated constraint violation handling for dynamic service composition. In *SCC '09: Proceedings of the 2009 IEEE International Conference on Services Computing*. IEEE Computer Society, Washington, DC, USA, 168–175.

WASSERKRUG, S., GAL, A., ETZION, O., AND TURCHIN, Y. 2008. Complex event processing over uncertain data. In *Proceedings of the second international conference on Distributed event-based systems*. DEBS '08. ACM, New York, NY, USA, 253–264.

WSBPEL. 2005. Wsbpel. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

WU, G., WEI, J., AND HUANG, T. 2009. Towards self-healing web services composition. In *Internetware '09: Proceedings of the First Asia-Pacific Symposium on Internetware*. ACM, New York, NY, USA, 1–5.

XIA, H. AND YOSHIDA, T. 2007. Web service recommendation with ontology-based similarity measure. In *Proceedings of the Second International Conference on Innovative Computing, Informatio and Control*. ICICIC '07. IEEE Computer Society, Washington, DC, USA, 412–.

YAHYAOUI, H., MAAMAR, Z., AND BOUKADI, K. 2010. A framework to coordinate web services in composition scenarios. *Int. J. Web Grid Serv. 6,* 2, 95–123.

YAO, D., LU, B., FU, F., AND JI, Y. 2010. A risk assessment algorithm based on utility theory. In *Proceedings of the Advanced intelligent computing theories and applications, and 6th international conference on Intelligent computing*. ICIC'10. Springer-Verlag, Berlin, Heidelberg, 572–579.

YEOM, G., TSAI, W.-T., BAI, X., AND LEE, Y. 2011. A design of policy-based composite web services qos monitoring system. *IJCCBS 2,* 1, 79–91.

YU, Q., LIU, X., BOUGUETTAYA, A., AND MEDJAHED, B. 2008. Deploying and managing web services: issues, solutions, and directions. *The VLDB Journal 17,* 3, 537–572.

YU, T., ZHANG, Y., AND LIN, K.-J. 2007. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web 1,* 1, 6.

ZAHOOR, E., PERRIN, O., AND GODART, C. 2010. Disc: A declarative framework for self-healing web services composition. In *Proceedings of the 2010 IEEE International Conference on Web Services*. ICWS '10. IEEE Computer Society, Washington, DC, USA, 25–33.

ZARRAS, A., VASSILIADIS, P., AND ISSARNY, V. 2004. Model-driven dependability analysis of webservices. In *Web services, International symposium on distributed objects and applications*. 69–79.

ZENG, L., BENATALLAH, B., DUMAS, M., KALAGNANAM, J., AND SHENG, Q. Z. 2003. Quality driven web services composition. In *Proceedings of the 12th international conference on World Wide Web*. WWW '03. ACM, New York, NY, USA, 411–421.

ZHAO, W. 2007. Bft-ws: A byzantine fault tolerance framework for web services. *Enterprise Distributed Object Computing Workshops, International Conference on 0*, 89–96.

ZHAO, W., ZHANG, H., AND CHAI, H. 2009. A lightweight fault tolerance framework for web services. *Web Intelli. and Agent Sys. 7*, 255–268.

ZHENG, Z. AND LYU, M. R. 2010. Collaborative reliability prediction for service-oriented systems. In *Proc. IEEE/ACM 32nd Int'l Conf. Software Engineering (ICSE'10)*. 35–44.

ABSTRACT

FAULT MANAGEMENT FOR SERVICE-ORIENTED SYSTEMS

by

AMAL ABEDALLAH ALHOSBAN

August 2013

Advisor: Dr. Zaki Malik

Major: Computer Science

Degree: Doctor of Philosophy

Service Oriented Architectures (SOAs) enable the automatic creation of business applications from independently developed and deployed Web services. As Web services are inherently unreliable, how to deliver reliable Web services composition over unreliable Web services is a significant and challenging problem. The process requires monitoring the system's behavior, determining when and why faults occur, and then applying fault prevention/recovery mechanisms to minimize the impact and/or recover from these faults. However, it is hard to apply a non-distributed management approach to SOA, since a manager needs to communicate with the different components through authentications. In SOA, a business process can terminate successfully if all services finish their work correctly through providing alternative plans in case of fault. However, the business process itself may encounter different faults because the fault may occur anywhere at any time due to SOA specifications.

In this work, we propose new fault management technique (FLEX) and we identify several improvements over existing techniques. First, existing techniques rely mainly on static information

while FLEX is based on dynamic information. Second, existing frameworks use a limited number of attributes; while we use all possible attributes by identify them as either required or optional. Third, FLEX reduces the comparison cost (time and space) by filtering out services at each level needed for evaluation. In general, FLEX is divided into two phases: Phase I, computes service reliability and utility, while in Phase II, runtime planning and evaluation. In Phase I, we assess the fault likelihood of the service using a combination of techniques (e.g., Hidden Marcov Model, Reputation, and Clustering). In Phase II, we build a recovery plan to execute in case of fault(s) and we calculate the overall system reliability based on the fault occurrence likelihoods assessed for all the services that are part of the current composition. FLEX is novel because it relies on key activities of the autonomic control loop (i.e., collect, analyze, decide, plan, and execute) to support dynamic management based on the changes of user requirements and QoS level. Our technique dynamically evaluates the performance of Web services based on their previous history and user requirements, assess the likelihood of fault occurrence, and uses the result to create (multiple) recovery plans. Moreover, we define a method to assess the overall system reliability by evaluating the performance of individual recovery plans, when invoked together.

The Experiment results show that our technique improves the service selection quality by selecting the services with the highest score and improves the overall system performance in comparison with existing works. In the future, we plan to investigate techniques for monitoring service-oriented systems and assess the online negotiation possibilities for combining different services to create high performance systems.

AUTOBIOGRAPHICAL STATEMENT

Amal Alhosban is a Ph.D. candidate in the Department of Computer Science at Wayne State University, received her Bachelor's degree in Computer Science from Yarmouk University and received her Master's degree in Computer Science from Al albayt University. She is a member of the Services COmputing REsearch (SCORE) Laboratory. Since starting her doctoral studies at Wayne State University she has served under various capacities (e.g. Graduate Research / Teaching Assistant, Student Assistant, and Part Time Faculty) in the Department of Computer Science. Her research interests include service computing, fault management, reliable distributed systems, semantic Web, data mining and wireless networks. She has published several research papers on these topics, and is an active member of ACM and IEEE.